Journal of Research and Practice in Information Technology

Special Issue on
Distributed Systems

Volume
33
number 2 2001

JOURNAL OF RESEARCH AND PRACTICE IN INFORMATION TECHNOLOGY
Formerly the Australian Computer Journal

The *Journal of Research and Practice in Information Technology* is a quarterly publication emphasising activities that have successfully connected fundamental and applied research in Information Technology with practical application. The journal publishes papers relating to both emerging research and to professional practice and thus contains articles that are of interest both to practicing information technology professionals and to university and industry researchers.

Editor (and address for correspondence):
Professor John F. Roddick
Editor-in-Chief
Journal of Research and Practice in Information Technology
Flinders University of South Australia
GPO Box 2100
Adelaide 5001, South Australia
Email: jrpit@cs.flinders.edu.au

Associate Editors
Associate Professor Graham Low
Associate Editor – Information Systems
The University of New South Wales

Professor Maria Orlowska
Associate Editor – Database Systems
The University of Queensland

Professor Bernhard Thalheim
Associate Editor – Conceptual Modelling
Brandenburg Technical University at Cottbus, Germany.

Dr Hugh Williams
Associate Editor – Computer Science
RMIT University

Denis Warne
Associate Editor – Software Engineering
BAE Systems

Michael Oudshoorn
Books Editor
The University of Adelaide

Professor Ian Witten
Associate Editor – Digital Libraries
University of Waikato, New Zealand.

Tom Worthington
Director, Publications Board
Australian National University

Office Bearers
J. Ridge, President
P. Kalkman, Vice President
P. Argy, Vice President
P. Ralston, Immediate Past President
R.G. Heinrich, National Treasurer
D.W. Furini, Chief Executive
PO Box Q534, QVB Post Office, Sydney, NSW 1230, Australia
Telephone: (02) 9299 3666
Facsimile: (02) 9299 3997
email: info@acs.org.au
URL: www.acs.org.au

Copyright© 2001, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by the Australian Computer Society Inc.

Print Post Approved PP255003/01660

Welcome to this special issue of the *Journal of Research and Practice in Information Technology.* The issue is devoted to the broad topic of distributed systems. This is indeed a broad area that is growing in importance both in terms of research activity and commercial adoption. The initial call for papers solicited papers that are consistent with the aims and objectives of the journal - namely the practice and practical application of research in distributed systems. The result is the collection of papers presented here that should be of interest to practicing information technology professionals as well as researchers from academia and industry.

Papers were submitted from around the world and all papers were reviewed by three international experts. The reviews provided valuable feedback to the authors and greatly aided to the selection of the papers for this special issue. The papers presented here reflect the diversity of the research taking place in distributed systems at the moment, and demonstrate that there is significant activity taking place in the field and that there is still much to be done.

The first paper, authored by Fatoohi, Gunwani, Wang and Zheng, examines middleware technology. They point out that there are a number of middleware technologies available at the moment including the Distributed Computing Environment (DCE), the Common Object Request Broker Architecture (CORBA), and the Distributed Component Object Model (DCOM). However, the underlying middle components are incompatible. Their paper examines the use of *DCOM-CORBA* and *DCE-CORBA* bridges to provide interoperability between different middleware environments.

The testing of distributed systems is notoriously difficult. The second and third papers of this issue address this important topic. Ghosh, Bawa, Craig and Kalgaonkar present an extensible and scalable framework called RiOT to allow the implementation of tools for test coverage measurement, test execution management and fault-based testing of distributed Java applications. The toolkit provides mutation operators to perform mutation analysis of interfaces and provides support for fault injection testing so that the fault-tolerance properties of the distributed application can be measured.

Huband and McDonald explore the use of Extensible Markup Language (XML) and XML schema as a means to specify a trace format for program employing the Message Passing Interface (MPI). MPI has been a significant contributor to the recent uptake and proliferation of parallel and distributed computation. This serves as the motivator for the provision for high quality debugging tools. Given that the application may be executing within a heterogeneous environment, it is important that the trace file formats are extensible, flexible and independent of architecture.

Williams and Parsons introduce the k-heterogeneous bulk synchronous parallel model as a means to develop general-purpose heterogeneous applications. They notice that it is not uncommon for a distributed application to execute across a collection of machines. These machines may differ in numerous ways including execution speed, data format, network connectivity, and memory resources. This may result in poor throughput as the low-end machines become the bottleneck of the system. Applicable environments include workstation clusters, the Internet, and computational grids. Williams and Parsons focus on the development of programs for a heterogeneous cluster of workstations.

The allocation of resources is critical for a distributed system to deliver optimal performance. Ravindran and Hegazy present a best effort resource allocation algorithm called RBA specifically aimed at asynchronous real-time distributed systems. Since the determination of optimal allocation strategies is computational intractable, RBA applies heuristics in an effort to derive a task allocation strategy that is as near optimal as practical. RBA proposes adaptation functions to describe the

anticipated application workload in the forthcoming time intervals. Heuristics are employed to determine the number of subtasks to be replicated at run-time in order to share workload and to maximise aggregate application benefit and minimise aggregate missed deadline ratio.

The final paper, authored by Balicki, also examines the performance of distributed systems through task assignment strategies. His approach is somewhat more theoretically based than that of the preceding paper. Balicki formulates the problem as a multiobjective combinatorial optimisation question, which is solved by an adaptive evolutionary algorithm.

I have enjoyed putting together this issue of the *Journal of Research and Practice in Information Technology*. It has afforded me the opportunity to showcase some of the research taking place in the area at the moment. I hope that you enjoy reading the papers and discovering the insight they provide.

Michael Oudshoorn
Guest Editor

Michael Oudshoorn is a Senior Lecturer in the Department of Computer Science at the University of Adelaide. His research interests include distributed systems and software engineering.

Performance Evaluation of Middleware Bridging Technologies

R. Fatoohi, V. Gunwani, Q. Wang and C. Zheng
Computer Engineering,
San Jose State University,
San Jose, California 95192, USA
Email: rfatoohi@email.sjsu.edu

In today's market, there are several middleware technologies, including: the Distributed Computing Environment (DCE), the Common Object Request Broker Architecture (CORBA), and the Distributed Component Object Model (DCOM). However, the underlying middleware components are incompatible; and therefore, a high-level heterogeneity problem has been created. Recently, software "bridges" have emerged that enable interoperability between different middleware environments. This paper presents the results of experiments to evaluate bridging technology using two DCOM-CORBA bridges and a DCE-CORBA bridge. Several configurations, depending on the number of machines and location of the bridge, are employed and two languages (C++ and Java) are used. The results show that the three bridges perform reasonably well for different configurations and language mappings.

Keywords: Middleware, CORBA, bridging, performance evaluation, interoperability.

CR Categories: C.2.4 — Client/server, C.4 — Measurement techniques, D.2.12 — Distributed objects.

1. INTRODUCTION

Many of today's organisations have a wide variety of computing systems that run different operating systems and software tools and interconnected by different networks. Applications tend to be even more diverse than the computing systems and networks they use. Some applications can only run on a single platform, while others run on several platforms. Middleware technologies exist to mask the underlying differences that may exist in platforms and applications. Middleware is defined as a set of common services that enable applications and end users to exchange information across networks. These services reside in the middle above the operation system and networking software and below the distributed applications (Bernstein, 1996).

In today's market, there are several middleware technologies, including: the Distributed Computing Environment (DCE) by The Open Group, the Common Object Request Broker Architecture (CORBA) by the Object Management Group (OMG), and the Distributed Component Object Model (DCOM) by Microsoft Corp. However, the underlying middleware components are incompatible; and therefore, a higher-level heterogeneity problem has been created.

Copyright© 2001, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 14 April, 2001, revised 2 August, 2001
Guest Editor: Michael Oudshoorn

The lack of interoperability of middleware technologies may cause many problems to organisations that have applications and services "belonging" to different middleware technologies. In an ideal world, developers write components and make them available to the rest of the world. End-users as well as other developers would use these components for their own work. However, if these components use different interfaces and middleware services, then they would be unable to communicate with each other. This means that once an organisation is "locked in" to a specific middleware technology, then it can only use components belonging to that technology. These components might not provide the best solution for any particular problem since there is no middleware technology that has all the answers to enterprise's problems.

Alternatively, these organisations may use software bridges that enable interoperability between different middleware environments. Basically, these bridges are processes that enable communication between clients in one middleware domain and servers in another domain. Bridging is not a new concept since it has been used for a long time in networking to interconnect dissimilar networking devices and communication protocols. By using bridges, the organisation would preserve the investment it made in its infrastructure and component development without rewriting its components for every new technology.

One of the main concerns in using bridges is the performance penalty that may incur since they add a layer of software between the communicating components. Here we have a typical case of tradeoff between performance and ease of use. The issue then is how severe the performance penalty would be to justify the use of the bridge. However, there is no comprehensive study of the performance and functionality of different bridging products.

This paper presents the results of experiments to evaluate the performance of several middleware bridging technologies. We focus on three middleware technologies: DCE, CORBA, and DCOM, which are briefly described in Section 2. Section 3 provides an overview of bridging technologies, followed by a brief description of two test problems. Next, the results of testing two DCOM-CORBA bridges and a DCE-CORBA bridge are presented and analyzed. Finally, we offer some concluding remarks.

2. MIDDLEWARE TECHNOLOGY

The section provides a brief description of three middleware technologies: DCE, CORBA, and DCOM; for more details, see Fatoohi (1998) and Orfali, Harkey and Edwards (1999).

DCE is a software infrastructure for developing distributed systems, produced by the Open Group (Rosenberry, Kenney and Fisher, 1992). DCE consists of a set of application programming interfaces and a set of run-time services. It is based on the RPC paradigm. In addition to its RPC, it provides a set of basic services that includes a naming service, a security and authentication service, and a time synchronisation service. DCE uses an Interface Definition Language (IDL) to specify procedure interfaces. DCE IDL is similar to other interface languages, including CORBA IDL, but it is unique (there is no standard for IDLs). Currently, only the C language is directly supported by the DCE IDL.

Object-Oriented DCE (OODCE) is an object-oriented wrapper of DCE developed by HP to support an object-oriented development paradigm (HP, 1994). In the OODCE development model, C++ is used as the primary language. OODCE uses the DCE IDL to define the server interface. It also uses an enhanced version of the DCE compiler, called idl++, to process the interface specification. OODCE is used in the core system of the NASA Earth Observing System Data Information System (EOSDIS). The EOSDIS is one of the largest information systems ever built that consists of legacy systems and data, variety of computers, high-speed networks and satellites.

CORBA is a standard for transparent communication between application objects developed by OMG (OMG, 1995). The CORBA Object Request Broker (ORB) enables clients and objects to communicate in a heterogeneous distributed environment. Using an ORB, a client can invoke a method on a server object transparently with respect to its location (local or remote) and its language implementation. The CORBA specification describes the interfaces and services that ORBs must have. The specification includes an IDL for describing interfaces. IDL mappings to several languages (such as C, C++, Java, Smalltalk, Ada, and COBOL) have been specified and provided by many vendors.

Distributed COM is an extension of the Component Object Model (COM), developed by Microsoft Corp. as an object-based programming model for developing and deploying software components (Eddon and Eddon, 1998). COM is the basis of many Microsoft object-based technologies such as ActiveX and OLE. COM has been part of MS Windows for some time. It separates object interfaces from their implementations, similarly to CORBA. COM defines a binary call standard for interfaces. It also defines a language for specifying interfaces called Microsoft IDL (MIDL). An MIDL compiler generates client proxies and server stubs in C, C++, or Java from an interface definition. The DCOM extension enables COM processes to run on different machines. It includes communication with remote objects, location transparency, and an interface to distributed security services.

3. BRIDGING TECHNOLOGY

In order to transparently couple components from different middleware technologies, some form of bridging software is needed. A bridge is therefore a process that allows a client in one middleware domain to make requests to, and receive replies from, a server in another middleware domain. The level of communication is determined by what kind of bridge is in use. A bridge possesses several properties, including unidirectional or bi-directional, static or dynamic, customized or commercial and other middleware-specific properties.

3.1 Unidirectional vs. Bi-directional

Unidirectional means that data can go in one direction only, either from the client to the server or from the server to the client. In order for data to go freely in either direction a bi-directional bridge is required. OMG coined the word inter-working to describe this behavior. This two-way interoperability allows functions to pass objects and to return objects. Combining two unidirectional bridges cannot accomplish the same deed since there is only one function call involved.

3.2 Static vs. Dynamic

A static bridge requires static proxy and stub implementation to perform marshalling for each interface. If there is any change to the interface, the proxy and stub must also reflect this change, thus the interface has to be recompiled to generate new proxy and stub codes to encode, decode and invoke the new interface. On the other hand, a dynamic bridge frees the user from this recompilation task since it has a generic proxy that is capable of marshalling all data types based on some form of a run-time type library. This type of information look-up can be costly in performance. Generally, dynamic bridges are not as efficient as static bridges, but with an optimisation like caching, dynamic bridges can catch up to static bridges in performance. Despite this potential performance factor, the generic mapping of the dynamic bridge does offer a smaller memory footprint than the interface-specific mapping of the static bridge.

Yet a third kind of bridges is on-demand bridges (Yang and Vogel, 1996). Here a bridge factory automatically generates the source code for a static bridge, usually based on the interface definitions of the client and server, which are to be bridged. Based on these definitions, bridging code must translate values between the two middleware domains, the mapping rules for the respective types are coded into the bridge factory.

3.3 Customised vs. Commercial

Customised bridge is any bridge that is not offered off the shelf. Normally, customised bridges are not recommended unless performance is critical or third party support is to be avoided. Otherwise, commercial bridge is a better choice for a lower cost in design, implementation, and maintenance. Nevertheless, if one must write his or her own bridge, the easiest approach is to create a Java-based bridge that takes advantage of the built-in support for COM in Microsoft's Java Virtual Machine, for example.

3.4 ORB-specific vs. ORB-neutral

This property is specific to CORBA but it is applicable to other middleware bridges. An ORB-specific bridge works only with the ORB products of the vendor who supplies the bridge. On the other hand, an ORB-neutral bridge can work with different ORBs.

4. TEST PROBLEMS

Two simple applications were developed to analyse the performance of different bridges. The first application, *count,* was designed to do a "ping" operation from the client to the server. In this application, which is similar to the Count program in Orfali and Harkey (1998), the client invokes the increment method one thousand times and measures the average round-trip time it takes (in milliseconds). The second application, *BankATM*, was designed to simulate a "bank" application, and used mainly for the interoperability study. It has several methods including: *deposit, withdraw, getBalance, and getAccount*. The performance of *count* was analyzed both with and without the bridges for comparison.

5. DCOM-CORBA BRIDGES

5.1 Overview

Objects using CORBA do not inter-work with objects that are based on the DCOM architecture, and vice versa. To meet this challenge, the OMG has created the DCOM-CORBA Inter-working specification (OMG, 1997a and OMG, 1997b). This specification has two parts: A and B. Part A defines the bi-directional mapping between non-distributed COM and CORBA. Part B extends the adopted part A specification to allow inter-working of DCOM and CORBA. As of CORBA 2.3, the inter-working specification has been incorporated into CORBA Architecture and Specification. The inter-working specification (parts A and B) specifies six configurations of the bridge, depending on the location of the bridge: on the client machine, on the server machine, or on a third machine (OMG, 1997a and OMG, 1997b), as shown in Figure 1. For more details concerning COM-CORBA bridges, see Geraghty *et al* (1999) and Rosen and Curtis (1998).

The location of the bridge determines the communication protocol between the client and the server. For example, when the DCOM client talks to the remote CORBA server and the bridge is on the client side, the Internet Inter-ORB Protocol (IIOP) is used for communication. On the other hand, when the CORBA client talks to the remote DCOM server and the bridge is on the client side,

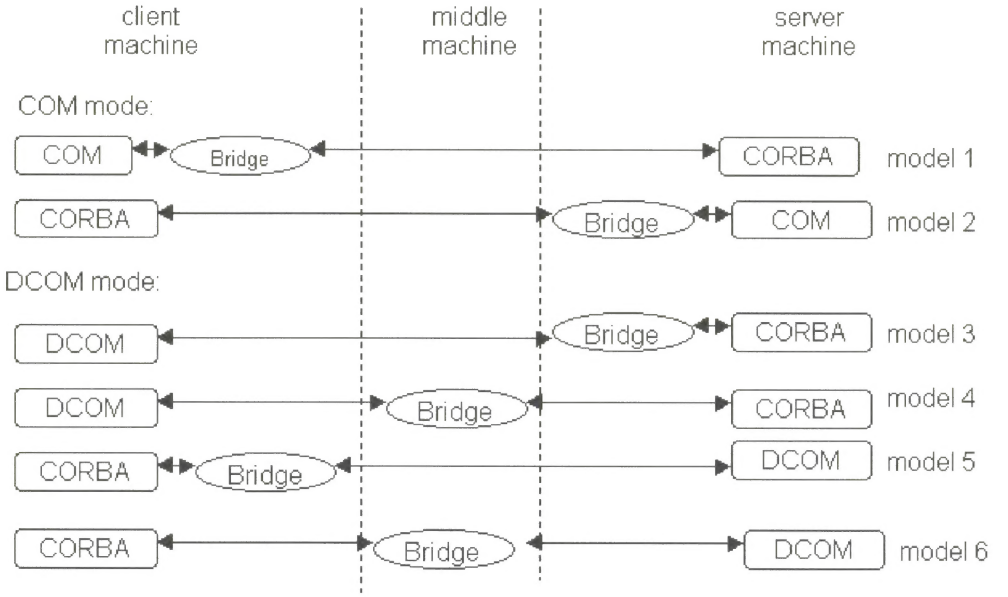


Figure 1. Bridge Configurations

Object RPC is used for communication. However, in practice, other factors determine the location of the bridge (Rosen and Cassady-Dorion, 1999). Housing the bridge with the client means that the actual bridging computation takes place on the client machine. It also implies that the inter-working software has to be installed on each client machine. Installing the bridge at the server side, on the other hand, means that the bridging computation moves to the server machine, where CPU cycles are in high demand. A third option is installing the bridge on a third machine. The disadvantage here is an additional process needs to be configured and managed.

5.2 Implementation and Results

Two DCOM-CORBA bridges were considered in this study: ObjectBridge by Visual Edge (Visual Edge, 1998) and OrbixCOMet by IONA Technologies (IONA, 1998). Both bridges are bi-directional and dynamic. ObjectBridge is also ORB-neutral while OrbixCOMet works with IONA Orbix3 only. Both of them are supposed to be compliant with the inter-working specification; however, we found that ObjectBridge might lack support for two of the specification configurations.

The *count* program was used to measure the performance of the two bridges. Each bridge was tested for all six configurations mentioned in the inter-working specification. The performance was measured with and without the bridge for comparison. Also, the impact of the language binding was considered. The *count* program was implemented with both C++ and Java for ObjectBridge. In addition, the client was also implemented with Java applet to measure performance of Web-based applications. In all cases, the same language was used for both the client and the server.

The DCOM-CORBA experiments were conducted using Windows NT 4.0 with Service Pack 5 running on Pentium II machines with 233 MHz and 128 Mbytes of memory. The machines are connected through an Ethernet 10Base-T network. Several software tools were needed for this study, including: ObjectBrident Enterprise Client version 1.1.1, VisiBroker for C++ version 3.3,

VisiBroker for Java version 3.4, Orbix3, OrbixCOMet, OrbixWeb version 3.1, MS Visual C++ version 6.0, MS Visual J++ version 6.0, SDK for Java version 4.0, DCOM Configuration, OLE-COM Object View, and SUN Java JDK 1.1.8 and JDK 1.2.2.

Tables 1 through 4 show the results (the average round-trip time for invoking the increment method 1000 times in milliseconds) for different configurations and language mappings. All data were collected when the network (Ethernet) is observed to be idle, usually at night. During performing the experiments, it was observed that network traffic could greatly affect the results.

5.3 Observations

The results for OrbixCOMet using Java were not available due to many obstacles in integrating Microsoft SDK for Java 4.0 with OrbixCOMet. Unlike ObjectBridge, OrbixCOMet does not work well with Microsoft SDK for Java. OrbixCOMet does not generate all the information that Microsoft SDK needs and vice versa. There are many missing pieces that make this task hard to accomplish.

The DCOM-CORBA results for C++ using both bridges (Tables 1 and 2) show that ObjectBridge is significantly faster than OrbixCOMet (by a factor of over three for most cases). In one of the configurations (CORBA client with DCOM server), the performance of OrbixCOMet is quite poor due to its current implementation. However, this configuration is not commonly used. These results also show that VisiBroker for C++ is faster than Orbix3 on a single machine as well as over the network (the upper left quadrants of Tables 1 and 2). The COM/DCOM results (the lower right quadrants) are, as expected, the same for both cases.

	CORBA Server	**DCOM Server**
CORBA Client	Client & server on one machine: **2.470 ms**	Client, bridge & server on one machine: **250.80 ms**
	Client & server on different machines: **2.664 ms**	Client & bridge on one machine, server on another: **250.87 ms**
		Client on one machine, bridge & server on another: **250.87 ms**
		Client, bridge & server on 3 different machines: **251.73 ms**
DCOM Client	Client, bridge & server on one machine: **3.021 ms**	Client & server on one machine: **0.23 ms**
	Client & bridge on one machine, server on another: **3.164 ms**	
	Client on one machine, bridge & server on another: **3.899 ms**	Client & server on different machines: **1.262 ms**
	Client, bridge & server on 3 different machines: **4.183 ms**	

Table 1. Results of OrbixCOMet C++ Server and Client

	CORBA Server	DCOM Server
CORBA Client	Client & server on one machine: **0.397 ms**	Client, bridge & server on one machine: **1.525 ms**
	Client & server on different machines: **0.991 ms**	Client & bridge on one machine, server on another: **1.604 ms**
		Client on one machine, bridge & server on another: **1.702 ms**
		Client, bridge & server on 3 different machines: **1.834 ms**
DCOM Client	Client, bridge & server on one machine: **0.841 ms**	Client & server on one machine: **0.23 ms**
	Client & bridge on one machine, server on another: **1.141 ms**	
	Client on one machine, bridge & server on another: **1.082 ms**	Client & server on different machines: **1.262 ms**
	Client, bridge & server on 3 different machines: **1.191 ms**	

Table 2. Results of ObjectBridge C++ Server and Client

	CORBA Server	DCOM Server
CORBA Client	Client & server on one machine: **1.432 ms**	Client, bridge & server on one machine: **2.113 ms**
	Client & server on different machines: **1.602 ms**	Client & bridge on one machine, server on another: **1.772 ms**
		Client on one machine, bridge & server on another: **1.842 ms**
		Client, bridge & server on 3 different machines: **1.983 ms**
DCOM Client	Client, bridge & server on one machine: **0.982 ms**	Client & server on one machine: **0.091 ms**
	Client & bridge on one machine, server on another: **1.332 ms**	
	Client on one machine, bridge & server on another: **1.352 ms**	Client & server on different machines: **1.922 ms**
	Client, bridge & server on 3 different machines: **1.402 ms**	

Table 3. Results of ObjectBridge Java Server and Client

	CORBA Server	DCOM Server
CORBA Client	Client & server on one machine: **1.573 ms**	Client, bridge & server on one machine: **2.383 ms**
	Client & server on different machines: **22.342 ms**	Client & bridge on one machine, server on another: **2.073 ms**
		Client on one machine, bridge & server on another: **22.451 ms**
		Client, bridge & server on 3 different machines: **22.885 ms**
DCOM Client	Client, bridge & server on one machine: **1.031 ms**	Client & server on one machine: **0.401 ms**
	Client & bridge on one machine, server on another: **1.372 ms**	
	Client on one machine, bridge & server on another: **1.392 ms**	Client & server on different machines: **1.973 ms**
	Client, bridge & server on 3 different machines: **1.472 ms**	

Table 4. Results of ObjectBridge Java Server and Applet Client

The single machine implementation shows that both bridges add an overhead of about 0.5 ms each (3.021 ms with the bridge compared to 2.664 ms without the bridge for OrbixCOMet, 0.841 ms with the bridge compared to 0.397 ms without the bridge for ObjectBridge). This is in comparison with CORBA-to-CORBA communication. In comparison with COM-to-COM communication, the bridge overhead is even higher.

The results (Tables 2 and 3) also show that Java performance is lower than C++ performance using ObjectBridge (as expected!). However, for COM-to-COM communication (with no bridging) on a single machine, Java is faster than C++. The Java COM server is an in-process server, housed in a DLL file while the C++ server is a local server (out-of-process) housed in an EXE file. But when an in-process server runs between machines, it has to run on top of a surrogate process, behaving more like an out-of-process server. Therefore, when the COM client and COM server are running on different machines, C++ outperforms Java.

The timing results for a DCOM client interacting with a CORBA server for multiple machine configurations are about the same regardless to the location of the bridge. These results are about up to 40% higher than the single machine configuration due to network overhead. Within the three different locations of the bridge (on the client machine, on the server machine, and on a third machine), the differences are insignificant.

Another observation is that the VisiBroker Java applet is very slow when the client and server are not running on the same machine (Table 4). VisiBroker requires Gatekeeper to be activated before an applet can run. Gatekeeper performs security check when the server and client are not running on the same machine.

6. DCE-CORBA BRIDGE

6.1 Overview

The DCE-CORBA Bridge is a commercial product developed by Inprise Corp. (Inprise, 1998). It can be used to enable CORBA clients to use IIOP to access data and transactions from DCE applications. The bridge acts as a standard CORBA server when accessed by CORBA applications. Bridge objects are CORBA objects that represent DCE servers. These objects are the means by which CORBA clients invoke operations on DCE servers. The bridge acts as a standard DCE client

	DCE Server	CORBA Server
DCE Client	Client & server on one machine: **0.698 ms**	Not supported
	Client & server on different machines: **1.089 ms**	Not supported
C++ CORBA Client	Client, bridge & server on one machine: **7.832 ms**	Client & server (C++) on one machine: **0.583 ms**
	Client & bridge on one machine, server on another: **7.996 ms**	
	Client on one machine, bridge & server on another: **7.856 ms**	Client & server (C++) on different machines: **0.842 ms**
	Client, bridge & server on 3 different machines: **7.475 ms**	
Java CORBA Client	Client, bridge & server on one machine: **8.334 ms**	Client & server (Java) on one machine: **1.472 ms**
	Client & bridge on one machine, server on another: **8.271 ms**	
	Client on one machine, bridge & server on another: **7.569 ms**	Client & server (Java) on different machines: **1.401 ms**
	Client, bridge & server on 3 different machines: **7.768 ms**	
Java applet CORBA Client	Client, bridge & server on one machine: **11.326 ms**	Client & server (Java) on one machine: **4.373 ms**
	Client & bridge on one machine, server on another: **10.909 ms**	
	Client on one machine, bridge & server on another: **28.161 ms**	Client & server (Java) on different machines: **25.807 ms**
	Client, bridge & server on 3 different machines: **29.661 ms**	

Table 5. Results of DCE-CORBA Bridge

when talking to any DCE server. Thus, bridge objects act simultaneously as CORBA servers and DCE clients. In its role as a CORBA server, the bridge encapsulates DCE systems in an object wrapper that enables the object-oriented CORBA clients to access DCE data and transactions using object semantics.

The Inprise DCE-CORBA bridge is a unidirectional, on-demand bridge. Only CORBA clients can access the DCE servers and not vice versa. The bridge is also an ORB-specific: it supports the VisiBroker ORB only. The bridge automatically generates a CORBA IDL file that maps to the DCE IDL. The generated CORBA IDL contains a CORBA object, called the bridge object, mapped to each DCE interface. The CORBA IDL can then be compiled with a CORBA IDL compiler to generate the client-side stubs and server-side skeletons. A CORBA client can then invoke methods on the bridge object.

The DCE-CORBA bridge can be used with standalone CORBA applications as clients or even with Web applications. In the case of Web applications, the CORBA client will be a Web browser with a Java applet embedded in the HTML page. The VisiBroker Gatekeeper is used to tunnel the HTTP requests (Inprise, 1998).

6.2 Implementation and Results

The count program was run for different configurations with and without the bridge for comparison. When running without the bridge, the program uses the DCE RPC mechanism in one case (DCE client, DCE server) and the CORBA ORB (CORBA client, CORBA server) in another. When the bridge is used, the client is a CORBA client and the server is a DCE server. The CORBA client accesses the bridge using the CORBA ORB and then the bridge invokes the DCE server using the RPC mechanism.

The DCE-CORBA experiments were conducted using Solaris 2.5.1 running on UltraSPARC IIi machines with 333 MHz and 128 Mbytes of memory. The machines are connected by a 10Base-T Ethernet network. Several software tools were needed for this study, including: Transarc DCE 1.1 for Solaris, VisiBroker for Java version 3.3, VisiBroker for C++ version 3.3, VisiBroker Gatekeeper, JDK 1.1.8 and GNU C and C++ compilers.

Tables 5 and 6 show the results (the average round-trip time for invoking the increment method 1000 times in milliseconds) of DCE-CORBA bridging for both DCE and OODCE, respectively, using different configurations. The DCE server was written in C while the OODCE server was written in C++. When a CORBA client talks to a CORBA server, both sides use the same language. All data were collected while the network (Ethernet) was idle.

6.3 Observations

The DCE timing results (Table 5) show that when the C++ CORBA client, bridge and DCE server are run on the same machine, the average time for the client to ping the server is 7.832 ms. This contrasts with an average time of 0.583 ms when a C++ CORBA client pings a C++ CORBA server. When a DCE client pings a DCE server, the average time was observed to be only 0.698 ms. Therefore, the extra overhead of around 7 ms incurred when connecting a CORBA client to the DCE server is attributable to the delay through the bridge.

When the client pings the server running on another machine but without the bridge, the average time increases due to network overhead. In the case of a C++ CORBA client pinging a C++ CORBA server, the average time rises to 0.842 ms from 0.583 ms in the single machine case. In the case of a DCE client pinging a remote DCE server, the average time is 1.089 ms as compared to 0.698 ms

	OODCE Server	CORBA Server
OODCE Client	Client & server on one machine: **0.797 ms**	Not supported
	Client & server on different machines: **1.127 ms**	Not supported
C++ CORBA Client	Client, bridge & server on one machine: **7.461 ms**	Client & server (C++) on one machine: **0.583 ms**
	Client & bridge on one machine, server on another: **8.041 ms**	
	Client on one machine, bridge & server on another: **7.510 ms**	Client & server (C++) on different machines: **0.842 ms**
	Client, bridge & server on 3 different machines: **7.805 ms**	
Java CORBA Client	Client, bridge & server on one machine: **7.847 ms**	Client & server (Java) on one machine: **1.472 ms**
	Client & bridge on one machine, server on another: **7.973 ms**	
	Client on one machine, bridge & server on another: **7.626 ms**	Client & server (Java) on different machines: **1.401 ms**
	Client, bridge & server on 3 different machines: **8.083 ms**	

Table 6. Results of OODCE-CORBA Bridge

in the single machine case. These results show that the CORBA communication (IIOP) is slightly faster than the DCE communication (RPC).

When comparing the C++ CORBA client to the Java CORBA client, we found that the average time to ping the DCE server rises from 7.832 ms to 8.334 ms. This indicates that the Java implementation is somewhat slower than the C++ implementation. This is also indicated by the fact that the average ping time for a Java CORBA client to Java CORBA server is 1.472 ms as opposed to 0.583 ms for the equivalent C++ case. When a Java applet is used as the CORBA client, the performance is significantly lower. The average ping time for a Java applet client to a DCE server is 11.326 ms and is 4.373 ms for a Java applet client to a Java CORBA server. The extra delay for the applet client can be attributed to the VisiBroker Gatekeeper, which is used to tunnel the HTTP request to the server.

The average times for a C++ CORBA client to ping a DCE server through the bridge and across the network range between 7.475 ms and 7.996 ms for the multiple machine configurations. Interestingly, these times are not much different from the value of 7.832 ms observed for the single machine case. This can be due to two reasons:

1) The time to transmit over the network is a small fraction of the overall time to connect the client and server through the bridge and,

2) With the client running on one machine and server on another, the CPU workload on both machines is lower than the single machine case and this may offset some of the overhead incurred by using the network.

Similarly, for the Java CORBA client to ping the DCE server, the average times range between 7.569 ms and 8.271 ms for the multiple machine configurations compared to 8.334 ms for the single machine configuration. Java applications are known to be CPU intensive and hence, the second reason given above may be even more applicable to this situation. The average ping time for the Java applet and DCE server case increases from about 11 ms for the single machine case to over 28 ms for two other configurations. The extra overhead is possibly due again to the VisiBroker Gatekeeper.

The OODCE timing results (Table 6) show that the performance of the bridge with DCE or OODCE on the server side is about the same (within about 5% difference). This shows that OODCE does not add a major overhead to DCE.

7. CONCLUDING REMARKS

In summary, the three bridges (Visual Edge ObjectBridge, IONA OrbixCOMet, and Inprise DCE-CORBA bridge) performed well using the two simple applications: count and BankATM. The BankATM application was used mainly to verify the proper operations of the bridges. Based on the results of the count program, ObjectBridge has a performance advantage over OrbixCOMet. However, due to the simplicity of the program, a more complex application should be employed to further verify this conclusion. Despite the performance issue, OrbixCOMet offers more bridging configurations and better documentation support.

The overhead incurred in using the Inprise DCE-CORBA bridge is approximately a few milliseconds for various client/server configurations. This is an acceptable overhead when leveraging legacy applications implemented using DCE or OODCE. An existing DCE framework can be utilised without any modification to the server code or the deployed configuration. However, unlike the two DCOM-CORBA bridges, the DCE-CORBA bridge is a unidirectional and on-demand bridge.

The three bridges performed reasonably well for the most common configurations (DCOM client with CORBA server and CORBA client with DCE server). The location of each bridge (close to the client, close to the server, or on a third machine) seems to have a little impact on the performance. In general, C++ implementations perform better than Java implementations and significantly better than Java applet implementations.

8. ACKNOWLEDGEMENT

We would like to thank Inprise, Visual Edge and IONA technologies for providing the three bridges for testing and evaluation. This project was partially funded through the Cooperative Agreement No. NCC2-1071 between NASA Ames Research Center and San Jose State University.

REFERENCES

BERNSTEIN, P. (1996): Middleware: A model of distributed systems services, *Communications of ACM,* 39(2): 86 – 98.

EDDON, G. and EDDON, H. (1998): Inside distributed COM. Microsoft Press.

FATOOHI, R. (1998): CORBA Framework for distributed virtual manufacturing, *Proc. 1998 World Automation Congress*, Anchorage, Alaska, USA, Robotic and Manufacturing Systems, 7: 597-602, TSI Press Series.

GERAPHTY, R., JOYCE, S., MORIARTY, T. and NOONE, G. (1999): COM-CORBA Interoperability, Prentice-Hall.

HP (HEWLETT PACKARD) (1994): HP object oriented DCE C++ Class Library Programmer's Guide, HP Co.

INPRISE (1998): Inprise DCE-CORBA Bridge User's Guide, Inprise Co.

IONA TECHNOLOGIES (1998): OrbixCOMet Desktop Programmer's Guide and References, Iona.
OMG (OBJECT MANAGEMENT GROUP) (1995): The common object request broker: Architecture and Specification, Revision 2.0.
OMG (1997a): COM/CORBA Inter-working Part A RTP Report, OMG Document orbos/97-09-07.
OMG (1997b): COM/CORBA Inter-working Part B Joint Revised Submission, OMG Document orbos/97-09-06.
ORFALI, R., and HARKEY, D. (1998): Client/server programming with Java and CORBA, 2nd ed., Wiley.
ORFALI, R., HARKEY, D. and EDWARDS, J. (1999): The essential client/server survival guide, 3rd ed., Wiley.
ROSEN, M. and CASSADY-DORION, L. (1999): How to connect with CORBA, *Enterprise Development,* 1(5): 22-32.
ROSEN, M. and CURTIS, D. (1998): Integrating CORBA and COM applications, Wiley.
ROSENBERRY, W., KENNEY, D. and FISHER, G. (1992): Understanding DCE, O'Reilly & Associates, Inc.
VISUAL EDGE (1998): ObjectBridge: The premier OLE/COM-CORBA interoperability solution, Visual Edge.
YANG, Z. and VOGEL, A. (1996): Achieving interoperability between CORBA and DCE applications using bridges, *Proc. International Conference on Distributed Processing,* 144-155.

BIOGRAPHICAL NOTES

Rod Fatoohi is an Associate Professor of Computer Engineering at San Jose State University. He holds a M.S. degree from Syracuse University and a Ph.D. degree from Old Dominion University. Before joining the academia, he worked for over eight years as a Research Scientist at NASA Ames Research Center. His research interests include distributed object technology, computer networking, and performance analysis. Dr Fatoohi is also a consultant with NASA and the IT industry and has published over thirty papers.

Vandana Gunwani, Qi Wang and Charlton Zheng were graduate students in the Computer Engineering department at San Jose State University.

An Extensible Framework for Testing Distributed Java Applications

Sudipto Ghosh, Nishant Bawa, Gerald Craig and Ketaki Kalgaonkar

Computer Science Department
Colorado State University
Fort Collins, CO, 80523, USA
Email: (ghosh, nishant, craigg, kalgaonk)@cs.colostate.edu

Testing applications involves the creation of test sets, their execution, management and adequacy assessment. An extensible framework, called RiOT, has been developed to allow the implementation of tools for test coverage measurement, test execution management and fault-based testing of distributed Java applications. Test coverage is measured in terms of elements covered in the interface description of an application's components. Mutation operators for interface mutation analysis are proposed and their implementation described. Fault injection testing for the assessment of the application's fault tolerance properties is made possible using the fault injection mechanism of the framework. An extension of the framework to allow test monitoring and control is described. The framework uses a hierarchical form of data communication between its different modules. The framework is designed to be scalable and easily extensible for future enhancements.

Keywords: Distributed applications, fault injection testing, Java RMI, Jiro, mutation testing, software testing, test adequacy criteria, test coverage, test management.

INTRODUCTION

Current software applications are highly complex. They are usually large-scale, distributed and heterogeneous, and have complex requirements. These factors increase the complexity of testing such applications. The state-of-the-art in software testing has not been able to keep pace with the developments in new distributed application technologies.

As part of a general testing process, a tester needs to develop test cases, execute them and observe failures. If there are failures, the cause is determined and necessary corrections are made. Given the infinite domain of inputs to an application, testing can never guarantee that the software is error free. At a practical level, testing is limited by the amount of resources available. Testing is only as good as the degree to which it "exercises" an application. A test suite that executes only a portion of an application with respect to a certain coverage domain, cannot be termed adequate with respect to that coverage domain. The effectiveness of testing performed is measured using a coverage metric (Zhu, Hall and May, 1997).

A test set is defined to be adequate with respect to some coverage metric if the test cases cover all the elements in the domain for that particular coverage metric. For example, if a test set executes

Copyright© 2001, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 15 April, 2001, revised 6 August, 2001
Associate Editor: Michael Oudshoorn

all the statements in a program, then the test set is adequate with respect to the statement coverage domain.

When an application is executed against a test set, the amount of coverage with respect to some domain can give an indication of the level of confidence in the tests being performed. This in turn determines the level of confidence in the tested product with respect to that coverage domain. Test adequacy criteria can also be used as stopping rules that determine when enough testing has been accomplished with respect to some coverage domain.

1.1 Coverage domains and test adequacy criteria

In this paper we identify a set of coverage domains for Java applications that can be used to define test adequacy criteria. We also describe a framework called RiOT, which can be used to measure these coverage metrics. RiOT stands for *Ri*gorous *O*bject *T*esting. We address the following issues that make testing of distributed applications challenging.

1. The lifespan of these applications is long which leads to an enormous amount of generated test data. Collection, management and processing of this data is a difficult task. RiOT provides a hierarchical set of data collection modules so that normal activities performed by the components under test are not unduly delayed.
2. Since applications have a large number of executing components, the adequacy criteria need to be scalable. In other words, with the addition of interfaces, the size of the coverage domain should not increase at a rate that makes it unmanageable for testing. In general, the larger the domains, the harder it is to develop test cases because of 1) the complexity involved, and 2) the number of test cases that are required. It is observed that certain adequacy criteria are difficult to satisfy because they require a large number of test cases. Given the resources available to the tester, it is not possible to develop and execute a large number of test cases. We define appropriate criteria that can be used for assessing the adequacy of distributed Java applications. These criteria are based on the descriptions of component interfaces instead of lower code level elements. We also define a fault-based criterion that uses a fault model based on interface descriptions.
3. Measuring performance characteristics of the application is important for tuning the performance of the components in it. RiOT can collect profiling metrics, such as, 1) number of times each method was executed, 2) number of times each exception was raised, and 3) total time required for a method to execute, (average, maximum and minimum latency).

Profiling provides insight into the areas of the code that are being exercised the most and the ones that are relatively under-utilised. Tests that are deficient with respect to method counts in important application areas can be modified to cover those areas more thoroughly.

1.2 Interface mutation testing

Using the interface description, we can define a fault-based testing strategy that creates mutants of original programs. This technique is based on mutation analysis as described by Demillo and Offutt (1991). Mutation operators model common programming errors and tests are developed to reveal these errors. We apply interface mutation analysis to assess the adequacy of test sets by requiring tests to distinguish mutants created using interface mutation operators.

1.3 Fault injection testing

Software applications used in environments critical to life and property need to have properties of high reliability, availability and fault tolerance. Such software applications usually have in-built

fault-handling or recovery modules. During system testing, faulty conditions are often not easily created and it is difficult to assess the fault tolerance of the applications. Fault injection testing is a technique that inserts faults into an application and ensures that the faults are triggered (Clark and Pradhan, 1995; Hsueh, Tsai and Iyer, 1997; Voas, 1999). Testers can then evaluate the quality of the fault recovery code.

An appropriate fault model is needed for fault injection testing and developing tools that inject faults. Distributed Java applications have different failure modes that are not found in single programs. Deadlocks between processes, network disconnects or component failures could cause problems with other components.

We describe a set of application-independent faults that can be injected in any distributed application. These "faults" simulate the effect of actual faults that may occur at the interfaces of the components.

1.4 Test management

A test management framework allows testers to visualise the entire application and obtain different dynamic views during testing. Views related to the dynamically changing architecture and tracing of method calls or messages between components or objects is useful for testers and maintainers. The framework should be able to handle large business applications. The impact on performance needs to be minimised. It is important to determine the requirements for data collection and analysis for an application. The architecture of the management system should be adequate to handle massive amounts of data collection without overloading the network and seriously affecting execution. It may be necessary to partition a large application's domain that allows for ease of management without information overload.

In the ideal case, the framework should be able to hook up with the application without requiring any extra effort. In most software development organisations users would prefer not to have to instrument and recompile code. Our goal is to have minimum "intrusion".

1.5 Organisation of this paper

The remainder of the paper is organised as follows. We describe the interface-based coverage metrics and associated test adequacy criteria in Section 2. Mutation operators are enumerated and the technique of interface mutation analysis is described in Section 3. The fault injection testing technique is explained in Section 4. Test monitoring and control is described in Section 5. The details of the operation of RiOT are given in Section 6. Finally, we summarise our work and outline directions for future work in Section 7.

2. TEST COVERAGE CRITERIA

For testing objects in applications, we assume that testers have already unit-tested individual methods. We are concerned with the testing of collections of objects that are present in server components. We identify coverage metrics defined on the basis of the interface exposed by a server component. A Java interface describes the signature of the methods and exceptions that can be thrown by the methods. The criteria that we propose are described in the following paragraphs.

2.1 Method coverage

A test set will lead to the execution of zero or more methods. A primary criterion of adequacy in testing a distributed application is invoking each method at least once. A test suite should exercise

all the methods available through the component's interface. We do not include private or protected methods that may be defined in a class. The only methods that we consider are the public methods as defined in the interface. A test set is considered adequate with respect to method coverage if 100% of the methods have been executed. We define method coverage at three levels. At the lowest level, the domain consists only of the methods defined in a single interface. At a higher level, we group the interfaces into a server component that implements the interfaces. Thus the coverage domain at this level is the union of all the methods in the interfaces implemented by the server component. At the highest level, we look at the union of all the methods defined in the interfaces of all the server components comprising the application. RiOT can measure coverage at all the three levels. At this point RiOT does not make a distinction between multiple instances of the same server component.

2.2 Exception coverage

It is a measure of the number of exceptions in an application that are raised when a test set is executed. A test set will lead to the execution of zero or more methods, which, in turn may raise zero or more exceptions. It is possible that more than one method can raise the same exception. The coverage domain of exceptions at the interface level consists of all the exceptions that can be raised by each and every method in that interface. A test set is adequate with respect to exception coverage if all the exceptions in that domain are raised as a result of some test in the test suite. At the component level, we group the interfaces into a server component that implements the interfaces. Thus the coverage domain at the component level is the union of all the exceptions that can be raised by methods in the interfaces implemented by the server component. At the highest level, we consider the union of all the exceptions defined in the interfaces of all the server components comprising the application. RiOT can measure coverage at all the three levels.

2.3 Method sequence coverage

A method sequence is a set of methods that are invoked one after the other in a definite order. A method sequence can be defined in at least two different ways.

1. Same object: For every object, if we consider the sequences of methods that are invoked on it, we get a coverage domain of method sequences.
2. Between objects: A different kind of sequence can be seen in collaboration and sequence diagrams in UML (UML, 2000) that show the interaction of several objects. There, a sequence is defined as a sequence of methods that are invoked by the set of interacting objects in response to one system operation.

Based on the methods declared in the interfaces, one can obtain a coverage domain of method sequences by applying both definitions. The coverage domains obtained this way consist of valid sequences only. One can define several invalid sequences as well. For an empty stack, a `push` followed by a `pop` is valid but a `pop` before a `push` is invalid. For method sequence coverage, we only consider the valid sequences.

Method sequence-based testing has been shown to be a powerful method for testing objects by Doong and Frankl (1994) in their work on testing Abstract Data Types. One problem with method sequence testing is that the total number of valid method sequences can be extremely large, possibly infinite, even for a small application. Hence achieving 100% method sequence coverage is an unrealistic task. A domain of method sequences is usually partitioned to make the size manageable.

2.4 Mutation score

A test set that obtains 100% method and exception coverage is usually less effective than tests adequate with respect to coverage criteria based on the code itself (Ghosh and Mathur, 2000). Experiments have shown that control and dataflow-based criteria are superior to method and exception coverage, but may require more test cases. Mutation analysis (Mathur, 1994) is a testing technique that assesses the quality of test sets by examining whether the test data can distinguish a set of alternate programs (mutants) from the original program. Mutants are created by applying a set of plausible faults one by one to the original program. The faults that are applied model common errors made by programmers. The faults are applied using mutation operators.

Suppose that there are M mutants created from a program out of which E are equivalent mutants, that is, they always produce the same output as the original program. If a test set is able to distinguish (*kill*) N mutants, then the mutation score is defined as $\frac{N}{M-E}$. A test suite is adequate if the mutation score is *1*.

Delamaro, Maldonado and Mathur (1996) proposed the idea of interface mutation for integration testing. Baudry, Hanh, Jezequel and Traon (2000) describe a mutation-based approach for evaluating trustable components. Ghosh and Mathur (2000) proposed the idea of interface mutation for object-oriented testing as a means to exercise more program paths by using "mutated" values of the method parameters. Since the paths executed in a method may depend on the values of parameters, mutating them usually results in the execution of different paths.

3. INTERFACE MUTATION ANALYSIS

RiOT allows the tester to generate mutants by specifying a set of standard mutation operators that can be applied to different methods. If there are m mutation operators, and n methods in an interface with each method having p parameters, then the number of mutants is $O(m * n * p)$.

The following mutation operators mutate the parameters of methods defined in the interface.

1. Increment a parameter by a constant (usually 1). This models off-by-one errors in the code where the parameter is used in some conditional evaluation in the server code.
2. Decrement a parameter by a constant (usually 1). This also models off-by-one errors in the code where the parameter is used in some conditional evaluation in the server code.
3. Swap parameters that have the same types. This models errors where a programmer passed parameters of the same type with a wrong order in the client code.
4. Set parameter to a specific value. This is based on the traditional mutation operator where a variable is set to boundary values in the domain of the variable.
5. Nullify references of objects being passed as parameters. This models errors where a null reference is not checked in the server code.

We will extend the design of RiOT to allow definition of other mutation operators. For example, one may define operators for container objects that cause the state of the objects to change by adding or removing elements in a container.

4. FAULT INJECTION TESTING

We simulate faults in the connections between components and faulty states in objects. Examples of faulty connections would include delays in the communication or links being cut off altogether. Effects of such faults may be delays in the response time of an object. Similarly if we wish to investigate the effect of an overloaded server from a client's perspective, we can introduce delays in the response time of a method call.

Since objects encapsulate data, it is not possible in most cases to access and alter their state attributes. Good programming practice precludes the availability of public attributes. Therefore, other means are required for putting objects in faulty states. Examples of faults introduced in objects could be:

1. Container object faults:
 (a) Make it empty
 (b) Remove an element, may be randomly
 (c) Add an element (default new element, or a clone of existing element)
 (d) Introduce faulty states in the contained object
2. Iterators over container objects:
 (a) Skip over elements
3. Input/Output Streams or Readers:
 (a) Skip bytes or characters read or written

5. TEST MONITORING AND CONTROL

In addition to monitoring coverage, testers may wish to monitor and control states of objects. The state information is useful in debugging applications. Capabilities are needed to monitor the events that relate to changes in state. Mathur, Ghosh, Govindarajan and Sridharan (1999) describe a facility that lets testers define events. Similarly, testers need to be able to define control actions for certain events. Actions may be applied at several levels: application, component, interface or method. Control actions could be of the form `start, suspend, wakeup` or `stop`. For example, one may `suspend` all method calls to a certain component and then "`wakeup`" the component. We are working on several issues in the implementation of such a feature. Section 6 explains the infrastructure that will allow us to monitor and control components.

6. RiOT

RiOT is a prototype implementation of a framework for testing distributed Java applications. RiOT provides the following features for testers. Testers can:

1. Define the application under test in terms of its components, interfaces and machines where the application would be executed. This information is referred to as a *Project configuration* or simply *Project*.
2. Save the application configuration information so that it can retrieved for retesting.
3. Specify the coverage domains related to component interfaces.
4. Obtain a static hierarchical view of components, interfaces, methods and exceptions.
5. Obtain coverage and profiling information about the methods and exception at the three levels in the hierarchy.
6. Perform interface mutation testing.
7. Perform fault injection testing.
8. Invoke different method sequences on the components.
9. Monitor and control the state of the components and objects in a component.

RiOT has been developed using Java. The communication architecture of RiOT has been implemented using Java RMI. The interceptors use *Dynamic Proxies* available in the JDK1.3. The monitoring and controlling mechanism is implemented using Jiro. Figure 1 shows the architecture of RiOT.

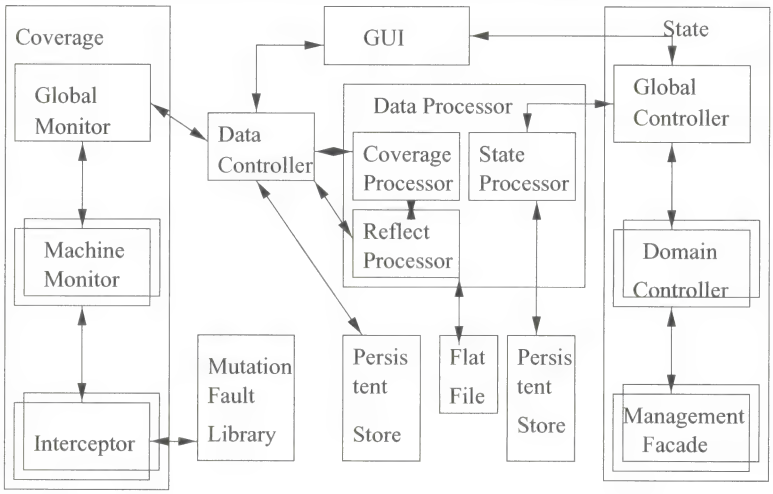


Figure 1: RiOT Architecture

The design of the RiOT follows a three-tiered architecture. The top tier consists of the GUI which allows a user to interact with RiOT. The second tier consists of five packages which form the main application logic for RiOT: 1) data controller package, 2) data processor package, 3) coverage package, 4) state package, and 5) mutation and fault library. The third tier consists of persistent storage.

6.1 Data controller package

The Data Controller package contains a Data Controller that is responsible for receiving requests from the GUI and taking appropriate actions. The Data Controller creates an instance of the Coverage Processor and the Global Monitor.

The Data Controller takes the information about the Project entered by the tester on the GUI and passes it to the Reflect Processor in the Data Processor package. The Data Controller controls the flow of data from the Global Monitor in the Coverage package to the Coverage Processor in the Data Processor package, and from the Coverage Processor back to the GUI. The Data Controller updates the data displayed on the GUI after a specified time interval. It also processes "on-demand" requests for updates via the refresh buttons on the GUI.

6.2 Data processor package

The Data Processor Package has the Coverage Processor, the Reflect Processor and the State Processor. The project information is directly passed to the Reflect Processor from the GUI. The Reflect Processor uses Java's reflection mechanism to extract the names of the public methods exposed by the interfaces, their parameters, and their corresponding exception names. This information is sent back to the GUI via the Data Controller to display the different constituents of the project in a tree, and also for mutation analysis.

The Reflect Processor creates an internal hierarchical representation of the project data which is passed to the Coverage Processor. The Reflect Processor is responsible for storing project information into persistent storage so that it can be retrieved in case the same project is needed for

retesting. The coverage and profiling information of the methods and exceptions is accumulated by the Global Monitor and conveyed to the Coverage Processor via Data Controller. The Coverage Processor is responsible for calculating the following statistics for the various observation levels:

1. Names of methods and exceptions covered at the interface, component or project level. From this information, coverage information at different levels is calculated.
2. Number of times a particular method or exception is covered and average, maximum and minimum method execution latency. This profiling information is only available at the interface level.
3. Time when a particular exception is raised.

All observations and calculations are returned to the GUI via the Data Controller package for display in pie charts, bar charts or lists.

The State Processor gets the state information regarding the components from the global controller. It processes this information and stores it on persistent storage. It also conveys the processed information to the Global Controller which sends it to the GUI for display.

6.3 Coverage package

The *Coverage Package* is responsible for accumulating coverage and profiling information about methods and exceptions collected from the interceptors of different components. This package consists of a Global Monitor, Machine Monitor and Interceptors. The communication link between them has been implemented using Java RMI to facilitate data flow between components distributed on different machines.

6.3.1 Global monitor

During the set-up of the project, the tester specifies the machines on which the Machine Monitors are to be started. This information is passed to the Global Monitor via the Data Controller. The Global Monitor creates instances of the Machine monitors and registers each with the RMI registry. During testing, the Data Controller prompts the Global Monitor for coverage information after a specified interval of time or whenever the tester specifically asks for an update. In response, the Global Monitor looks up each Machine Monitor. It obtains data regarding coverage and profiling information of methods and exceptions from each Machine Monitor. The data obtained from all the Machine Monitors is then passed to the Data Processor via the Data Controller.

6.3.2 Machine monitors

These monitors accumulate coverage and profiling data from of all the components that reside on the same host. The data collected by the Machine Monitors is updated by the interceptors of the components. Upon request by the Global Monitor, this data is passed to it for inclusion in the global data containers.

6.3.3 Interceptors

To find coverage information, we need to know which methods were invoked and which exceptions were thrown. We can determine when a method was invoked if we are able to intercept that method call. For applications built using CORBA, interceptors are available to intercept the calls either at the client side or at the server component side (Vogel and Duddy, 1998). Since RMI does not have an interceptor-like mechanism, we use dynamic proxies provided in the JDK1.3 reflection API to implement a similar interception mechanism. Figure 2 shows how dynamic proxies can intercept a method call.

A proxy is a surrogate or delegate for a component. All method calls to a component are passed through the proxy. Each proxy instance has an associated invocation handler. When a client invokes

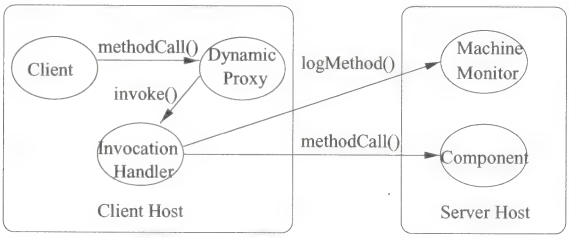


Figure 2: Method Call Interception

a method call on the component, the call is sent to the proxy of the respective component. The proxy in turn dispatches the call to the `invoke` method of the invocation handler. The test client that is interested in a component, first looks up the component using RMI and gets a reference to the component. Using this reference the client creates an instance of a dynamic proxy and then invokes the method on the proxy instead of the reference returned by RMI. The proxy directs the calls to the `invoke` method of the invocation handler, which in turn directs them to the actual component. The proxy and invocation handler operate in the client address space.

On each host there are Machine Monitors which log information about method calls invoked and exceptions thrown by the components that reside on that host. The invocation handler looks up the Machine Monitor of the host where it resides. The data collected by the invocation handler is logged by the Machine Monitor and ultimately collected by the Global Monitor.

In the invoke method of the invocation handler we extract the following information regarding the method call:

1. Component, interface and method names of the call.
2. Time when the method call was invoked and when it returned.

Exceptions that are thrown in response to the remote method invocation on the component are caught by the invocation handler. The following information regarding the exception is extracted and logged in the Machine Monitor.

1. Component, interface and method name that raised the exception.
2. The name of the exception thrown.
3. The time when the exception was thrown.

Our communication mechanism eases the dependence of the Interceptors on one Global Monitor. In case all the Interceptors communicated with a single Global Monitor, it would get overburdened. This would cause delays in the Global Monitors' response to the Interceptors. To avoid this situation, the Interceptors communicate with the Machine Monitors. Since there is one Machine Monitor for each machine, it is less likely to be overwhelmed and will respond faster to the Interceptors. Hence, the delegation of communication tasks to the machine monitors results in better performance.

6.4 Mutation and fault library

Mutation testing is done at the method level. The tester selects the methods and the mutation operators to be applied. This information consists of the component name, interface name, method name, mutation operator, the index (or indices) of parameters and any specific value to be used for the mutation. The information provided is passed to all the Machine monitors which forward the necessary information to the interceptors. The interceptors have access to a mutation operator library. When a method call that is supposed to be mutated appears at the interceptor, the parameter

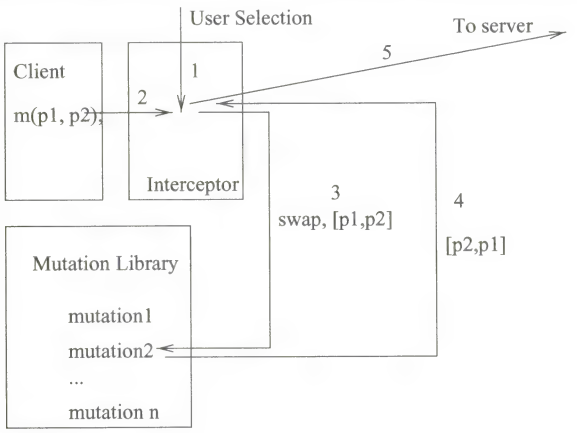


Figure 3: Interface Mutation in RiOT

array is passed to the mutation library along with the mutation operator information. The parameter array is changed appropriately and then propagated back to the Invocation Handler. The change in one or more parameters causes the mutation to occur. The mutation library can be easily extended to apply more operators. Figure 3 illustrates how the parameters are mutated.

Fault injection is applied in exactly the same manner. The fault library includes `Fault` classes such as `delays`. Work is in progress to include other fault types.

6.5 State package

The State Package has been designed but not fully implemented. The State Package comprises the Global Controller, various Domain Controllers and Management Facades. The tester specifies the components to be monitored and also how to change the state of the components. This information is sent from the GUI to the Global Controller. The Global Controller checks the domain name in which the targeted component resides and forwards the state change request to the Domain Controller of that particular domain. The request can be of two types:

1. To get the state information of a component.
2. To change the state of a component.

The Global Controller gathers information regarding states of components and the entire application and sends it to the State Processor. The State Processor processes the information, stores it in persistent storage, and passes the processed information to the Global Controller. The Global Controller sends the appropriate information to the GUI for display. Under a Global Controller, there may be several Domain Controllers.

A domain is a subsection of the entire distributed application. Jiro requires dividing a large application into domains for ease of management. The Domain Controller is responsible for maintaining state information about all the Components in that particular domain. In response to a state information request, it sends the information to the Global Controller, which in turn sends it to the State Processor.

If the request is to change the state of the component, the domain controller configures the Management Facade of the target component so that the Management Facade alters the state of the component as desired by the tester.

6.5.1 Management facade and the event service

The Management Facade is a manifestation of the Federated Beans Architecture (Sun Microsystems 2001a, 2001b). The facade acts as an interface to the components. Management Facades are employed by RiOT to monitor and control the state of components. RiOT leverages Jiro's event service for monitoring. An event service is a collection of topics.

The event source post events to a particular topic, which in turn delivers the event to the event subscribers. For each event source, there is a corresponding topic to which the event is posted. For a component C there is a topic T such that all the events corresponding to the state change of component C are posted to topic T. Topic T in turn delivers the event to the subscriber of that topic. An event is an object that contains information about some object's state change that other objects are interested in knowing. The subscriber objects need to subscribe for a particular topic with the event service via a lease. The lease is only valid for some period of time and needs to be renewed after that period is over. There is one event service in each domain. An event service in one domain can be a subscriber to an event service of some other domain. This allows an event to be propagated between the event services of different domains.

Figure 4 describes how we plan to use the Jiro event service. Whenever the state of a component C_{11} changes, it notifies its corresponding management facade MF_{11}. MF_{11} generates an event object and posts it to a topic T_{11}. T_{11} posts the event to the Domain Controller DC_1 of the domain in which the component resides. The Domain Controller is a subscriber to all the topics of the event service in that domain. Hence all the events generated by the Components in that domain are delivered to the Domain Controller.

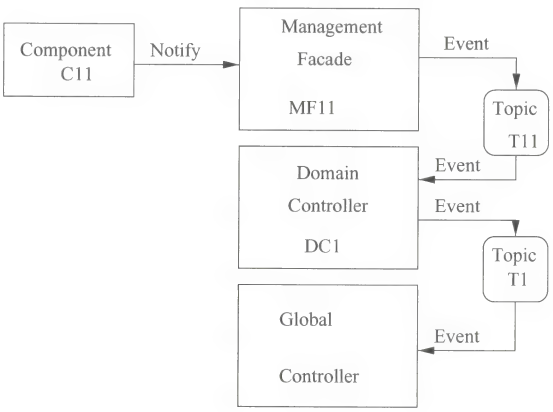


Figure 4: Using Jiro's Event Service

There is also a T_1 corresponding to each of the Domain Controllers. The Global Controller subscribes to each of these domain topics. Whenever an event is delivered to a Domain Controller, it posts the event to the domain topic of that domain. All the domain topics in turn deliver the events to the Global Controller. When the event reaches the Global Controller it extracts information regarding the name, domain and state change of the source component which generated the event. This information is then passed on to the State Processor.

6.5.2 Controlling States of Components

When the tester makes a request to change the state of a component, the request is delivered to the Global Controller. It forwards the request to the Domain Controller of the domain in which the

target Component resides. The Domain Controller directs the Management Facade of the target component to change the state of the component as desired by the tester.

6.6 Storage

The third tier of RiOT's architecture consists of persistent storage facilities including a database and flat files. Information regarding the project set up is saved in a flat file. Coverage related information is stored in a DBMS.

7. SUMMARY AND FUTURE WORK

This paper listed the issues in testing distributed applications that were addressed by RiOT. Test adequacy criteria based on elements in the description of Java interfaces of objects were defined. The criteria are based on method, exception, method sequence coverage and interface mutation score. RiOT's coverage measurement and fault-based testing techniques were described. The fault-based techniques are interface mutation analysis and fault injection testing applied to distributed Java applications. A set of mutation operators and a set of generic faults that can be injected were defined.

RiOT can be easily extended to add new functionality. This is possible because of the communication hierarchy, the dynamic proxy mechanisms used, and the components used to perform management. One can always define new coverage domains based on the interface. The monitoring framework will be able to collect the new coverage metrics. New mutation operators and faults can be easily added to the mutation library as long as the operators and faults are based on the interface description. Once RiOT has the capability of defining new states, one will be able to add monitors and controllers using the state package to manage these states.

RiOT is a research prototype that will be used to empirically evaluate the effectiveness of the proposed coverage criteria and fault-based testing techniques. An evaluation of the criteria will give guidance to testers in selecting appropriate criteria based on resources available and the criticality of the application. Work also needs to be done to identify appropriate method sequences. Sequence coverage is one coverage domain that is unlikely to be scalable. However, tests using sequence coverage criteria are more likely to reveal errors. Empirical studies of these criteria needs to be done for different application domains. The effectiveness of the criteria on asynchronous and concurrent execution needs to be evaluated.

The framework will be enhanced to allow the injection of arbitrary faults that are possibly application dependent. More mutation operators based on the types of objects will be implemented. The state control mechanism needs to be implemented and work needs to be done to define more states in addition to the ones mentioned in this paper.

REFERENCES

BAUDRY, B., HANH, V. L., JEZEQUEL, J. M. and TRAON, Y. L. (2000): Trustable components: Yet another mutation-based approach. *Proceedings of Mutation 2000, Kluwer Academic Publishers,* 47-54.

CLARK, J. A. and PRADHAN, Y. K. (1995): Fault injection: A method for validating computer-system reliability. *IEEE Computer* 28(6):47—56.

DELAMARO, M. E., MALDONADO, J. C. and MATHUR, A. P. (1996): Integration testing using interface mutation. In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE '96),* 112—121.

DEMILLO, R. A. and OFFUTT, A. J. (1991): Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 17(9):900-910.

DOONG, R. K. and FRANKL, P. G. (1994): The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology* 3(2):101—130.

GHOSH, S. and MATHUR, A. P. (2000):"Interface mutation". In *Proceedings of MUTATION 2000* 112—123, San Jose California.

HSUEH, M. C., TSAI, T. K. and IYER, R. K (1997): Fault injection techniques and tools. *IEEE Computer* 30(4):75—82.

MATHUR, A. P. (1994): "Mutation testing", volume 1 of *Encyclopaedia of Software Engineering,* MARCINIAK, J. (ed.). Wiley Interscience.

MATHUR, A. P., GHOSH, S., GOVINDARAJAN, P. and SRIDHARAN, B. (1999): A framework for assessing test adequacy, architecture extraction, metering, monitoring and controlling distributed component-based systems. In *Proceedings of 1st Symposium of Reusable Architectures and Components for Developing Distributed Information Systems,* Orlando, Florida, August 1999.

SUN MICROSYSTEMS (2001a): Federated management architecture.
URL http://www.sun.com/aboutJava/communityprocess/final/jsr009.

SUN MICROSYSTEMS (2001b): Jiro Technology.
URL http://www.jiro.com/.

THE OBJECT MANAGEMENT GROUP (OMG) (2000):Unified modeling language Version 1.3, OMG. URL http://www.omg.org.

VOAS, J. M. (1999): Certifying software for high-assurance environments. *IEEE Software* 16(4):48-54.

VOGEL, A. and DUDDY, K. (1998): *Java programming with CORBA.* John Wiley and Sons, USA.

ZHU, H., HALL, P. A. V. and MAY, J. H. R. (1997):Software unit test coverage and adequacy. *ACM Computing Surveys* 29(4):366-427.

BIOGRAPHICAL NOTES

Sudipto Ghosh received the Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of Technology, Kanpur, India in 1993. He received the Master of Science degree in Computer Science from Iowa State University in 1995 and the Doctor of Philosophy degree from Purdue University in 2000. He is currently an Assistant Professor in the Computer Science Department at Colorado State University, Fort Collins, Colorado. His research interests include software engineering and distributed object systems. He is a member of the Association of Computing Machinery and the IEEE Computer Society.

Nishant Bawa received the Bachelor of Engineering degree in Electronics and Telecommunication from the University of Pune, India in 1998. He is currently pursuing a Master of Science degree in Computer Science at Colorado State University, Fort Collins, Colorado. His research interests include distributed and component-based systems. He is currently a graduate assistant at Colorado State University.

Gerald Craig is completing requirements for a Master of Science degree in Computer Science at Colorado State University. His areas of interest include software engineering and testing of distributed systems. He is currently employed part-time as a testing engineer with Hewlett Packard's Manageabilty Solutions Laboratory in Fort Collins, Colorado.

Ketaki Kalgaonkar received the Bachelor of Engineering degree in Computers from the University of Poona, India in 1999. She is currently pursuing a Master of Science degree at Colorado State University, Fort Collins, Colorado. Her research interests include software engineering and testing distributed software. She is currently a graduate assistant at Colorado State University.

Using XML to Specify a Trace Format for MPI Programs

Simon Huband and Chris McDonald

Department of Computer Science & Software Engineering
The University of Western Australia
Crawley, Western Australia 6009, Australia
{huey, chris}@cs.uwa.edu.au

Trace files have long been used to assist correctness debugging and performance debugging of parallel programs. With the advent of implementations of the Message Passing Interface (MPI) standard, parallel and distributed computing has become more common, and thus the need for quality debugging tools has increased. It is important that trace file formats be extensible, flexible, and architecturally independent, the latter particularly if analysis is performed on a different platform to that which generated the trace. In this paper we propose a set of requirements for MPI-based trace libraries, and present a preliminary trace library, `tracempi`*. An important contribution is that this trace format uses the Extensible Markup Language (XML), and XML Schema. By doing so, it is architecturally independent, well defined, and easily extended.*

Keywords: MPI, parallel program debugging, XML, trace files
CR Categories: D.1.3 [Concurrent Programming], D.2.5 [Testing and Debugging]

1 INTRODUCTION

By drawing upon multiple processors concurrently, parallel and distributed processing allows us to address computationally intensive tasks for which the use of a single processor is considered inadequate. Some application domains, such as weather forecasting and climate research, demand the power afforded by parallel and distributed processing. Finer grained tasks, such as parallel matrix manipulations and parallel sorting, serve as an important basis for more complicated parallel programs. Across many levels of computing, parallel and distributed processing is an important asset.

With the advent of popular message passing environments such as the Parallel Virtual Machine (PVM) (Geist, Beguelin, Dongarra, Jiang, Manchek, and Sunderam, 1994), the Message Passing Interface (MPI) standard (Message Passing Interface Forum, 1995), and later MPI-2 (Message Passing Interface Forum, 1997), parallel programming has become increasingly standardised and portable. Implementations of MPI exist for architectures ranging from dedicated supercomputers through loosely connected networks of heterogeneous workstations. Parallel programming has become increasingly accessible and thereby increasingly popular.

However, designing and debugging parallel programs remains difficult. Poorly designed parallel programs perform sub-optimally and, compared to sequential programs, are more difficult to code,

Copyright© 2001, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 14 April, 2001, revised 9 August, 2001
Associate Editor: Michael Oudshoorn

more error prone, and harder to debug. The problem with parallel programming is the added difficulty of understanding and assimilating the expected and actual behaviour of multiple interacting processes.

Debugging parallel programs is notoriously difficult, for reasons that are well known, well-understood, and very difficult to address (McDowell and Helmbold, 1989). Unlike most sequential programs, debugging usually has two meanings when referring to parallel programs: correctness debugging, and performance debugging. Correctness debugging, often the only debugging applied to sequential programs, is the identification and correction of semantic errors. Performance debugging refers to the identification and improvement of aspects of the program that cause it to perform sub-optimally. It is usually necessary to perform correctness debugging prior to performance debugging.

Whilst various techniques can be applied to parallel program debugging, performance debugging invariably involves some form of tracing. Tracing parallel program execution involves logging accesses to parallel library routines, in particular those involving inter-process communication. For performance debugging, additional log data is often routinely accumulated, such as records of processor utilisation and memory usage. Whilst traces can be analysed during execution, they are typically stored on disk for post-mortem analysis. Trace files are used for performance analysis in a number of ways, including statistically, visually, and even for performance prediction.

Program traces are also used for correctness debugging. The complexity of parallel programs is mostly due to inter-process communication. Tracing is a manner of observing this inter-process interaction. With sufficient data trace files can be used to replay programs (Ronsse, De Bosschere, and Chassin de Kergommeaux, 2000) in order to force repeatable behaviour, or they could be analysed for logical errors (Kranzlmüller, 2000), such as isolated receive events, or error-prone nondeterministic behaviour.

As trace files can be helpful in so many ways, they are a valuable asset to the parallel programmer. The main drawback with trace files is that they can be very large. Trace files are usually analysed via additional software, which runs external to, and often after the termination of, the parallel program being considered.

To assist trace library development, the MPI standard includes a profiling interface that enables developers to neatly construct portable trace (or profiling) libraries. Consequently there are many MPI trace libraries, although not all use the recommended profiling interface. Most libraries are only appropriate for performance analysis since they expect clean program termination, and choose not to trace information useful for correctness debugging.

Moreover, existing trace libraries usually use custom trace file formats, with custom syntaxes that are not always well documented. Consequently, often only programs developed in conjunction with the trace library actually use that trace library. Whilst custom formats introduce some efficiencies, they often do so by sacrificing extensibility and usability. Furthermore, existing trace libraries are custom developed for specific problem domains, and thus implement only subsets of the possible functionality that full tracing can offer.

This paper outlines the fundamental features that trace libraries should offer in order to support both correctness and performance debugging. It then presents a new MPI trace library, `tracempi`, which represents a partial fulfilment of the trace library requirements. A significant part of this research involves the decision to adopt XML (the Extensible Markup Language) (Bray, Paoli, Sperberg-McQueen, and Maler, 2000) and XML Schema (Fallside, 2001) as the trace file format.

XML is a portable, well-understood, well-supported, meta-markup language used in industries ranging from aeronautics and physics to hospitality and travel.

In addition, it is also important that the tracing of all possible data be allowed. Tracing everything has the affect of not constraining one's view on how debugging should be performed. Without a restriction on the available data, there is no restriction on what a developer can choose to do with it. Moreover, by using XML, the infrastructure for parsing a trace is already there. These factors will encourage developers to develop their own tools that employ this format, and thus encourage the growth of a diverse family of trace file based visualisers, debuggers, and analysers.

In the next section some existing trace libraries for MPI are described, followed in Section 3 by a discussion of the general requirements that trace libraries should satisfy. In Section 4 the advantage of using XML and XML Schema is described, and then, in Section 5, the XML-based trace library `tracempi` is presented. Finally, Section 6 concludes this paper.

2 RELATED WORK

In this section we describe parallel programming environments that include MPI trace libraries. Most existing libraries are purely intended for performance-based debugging, thus limiting their use with respect to correctness debugging. We first describe trace libraries belonging to environments that are still under development. The development of most of the environments discussed began in the early 1990s.

2.1 `mpich`

The `mpich` implementation of MPI (Gropp, Lusk, Doss, and Skjellum, 1996), under continued development at the Laboratory for Advanced Numerical Software (Mathematics and Computer Science Division, Argonne National Laboratory), ships with three distinct profiling libraries, `ALOG`, `CLOG`, and `SLOG`. The `ALOG` format is an ASCII text format, `CLOG` is a binary (smaller), more flexible, updated version of `ALOG`, and `SLOG` is a scalable format designed such that the correct graphical display of traces does not require reading entire files (which may be very large). These libraries use the MPI profiling interface. `SLOG` actually consists of two different formats, `SLOG-1` (state-based logging format), and `SLOG-2` (drawable-based logging format). `SLOG-2` is a project under development.

Since they are used for performance analysis only, correct trace files are often not produced if program failure occurs during tracing. Jumpshot-2 (Zaki, Lusk, Gropp, and Swider, 1999) and upshot (Herrarte and Lusk, 1991), both shipped with `mpich`, use `CLOG` and `ALOG` trace files respectively.

2.2 Vampir

Vampir (Visualisation and Analysis of MPI Programs) (Nagel, Arnold, Weber, Hoppe, and Solchenbach, 1996) is a popular commercial tool developed by Pallas which is designed to provide versatile performance visualisation and analysis of MPI programs. Vampir reads trace files generated by Vampirtrace. Vampirtrace is designed to incur low overhead, and uses the MPI profiling interface.

At present Vampirtrace does not produce complete trace files if processes do not terminate normally, although a future release is planned to change this (Werner Krotz-Vogel, private communication).

2.3 Monitoring and Debugging Environment (MAD)

MAD (Kranzlmüller, Grabner, and Volkert, 1997) is an ongoing project at the Department for Graphics and Parallel Processing at Johannes Kepler University, Linz, Austria. It is designed to be an environment that assists the user in both error detection and performance tuning.

The Event Monitoring Utility (EMU) is the component responsible for trace file generation. Of the trace libraries discussed, it is the only library designed for both correctness and performance debugging. ATEMPT (A Tool for Event ManiPulaTion), also a part of MAD, is the visualisation tool used to apply both correctness and performance analysis techniques to trace files generated by EMU.

EMU traces only the necessary data for ATEMPT, including logical clock information, source code line numbers and file names, and timestamps. Trace events are also buffered, where the detection of abnormal program termination causes the buffers to be written to disk.

2.4 MPICL

MPICL, developed by Patrick H. Worley at Oak Ridge National Laboratory, is an MPI extension to the Portable Instrumented Communication Library (PICL) (Worley, 1992). PICL is a library that provides a message-passing interface on a variety of multiprocessors. PICL programs produce timestamped trace data, including processor busy/idle times, and simple user defined events. MPICL trace data uses the same ASCII text format as PICL, but is extended to handle the additional MPI routines.

MPICL trace files are displayed by the performance analysis program ParaGraph (Heath and Etheridge, 1991). MPICL is designed for performance analysis, and does not output traces reliably in the event of program failure. MPICL uses the MPI profiling interface, and is still being extended.

2.5 Pablo Performance Analysis Environment

The Pablo Research Group, a part of the Department of Computer Science at the University of Illinois at Urbana-Champaign, is actively developing tools for the performance analysis of parallel programs. One tool under active development is SvPablo (De Rose and Reed, 1999). As part of its functionality, SvPablo performance browsing involves correlating performance data against program source code. This performance data is stored in the Self-Defining Data Format (SDDF) (Aydt, 1992).

SDDF is a specification for a generic data file format. Prior to any particular data record instance occurring, a definition of the format of that data record (the data record descriptor) must first appear, so that subsequent parsing of data record instances can be handled. Thus it is up to the developer to decide what types of data records appear, whilst still being able to work with a well defined data file format. SDDF files can be in either an ASCII format (portable), or a binary format (not guaranteed to be portable). The SDDF format has been adopted as the standard PVM trace file syntax (Kohl and Geist, 1996).

One of the Pablo tools is an SDDF-based trace generator for MPI programs. It employs the profiling interface, and as with most performance analysis based trace libraries, events are buffered in memory prior to being written to disk.

2.6 Automated Instrumentation and Monitoring System (AIMS)

AIMS (Yan, Sarrukai, and Mehra, 1995) was developed at the NASA Ames Research Center under the sponsorship of the US High Performance Computing and Communication Program. The goal of

the project was to provide a suite of software tools for the measurement and analysis of the performance of Fortran and C message-passing programs using the NX, PVM, or MPI communication libraries. The AIMS project has been inactive for several years.

As part of the AIMS software suite, `xinstrument`, `monitor`, and (optionally) `pc` are used in conjunction to generate trace files that are analysed post-mortem by AIMS's visualisation and analysis toolkit.

The `xinstrument` tool is responsible for instrumenting source code prior to compilation and linking against the `monitor` library, which is responsible for the actual tracing. During execution, trace events are buffered, and only written to disk either at the conclusion of the program, or when buffer space is exceeded. The `pc` utility is used (post-mortem) to compensate for timing perturbations introduced by tracing.

2.7 Annai

The Annai integrated tool environment (Wylie and Endo, 1996), developed as part of the Joint CSCS/NEC Collaboration in Parallel Processing, includes tools for parallelisation, debugging, and performance monitoring and analysis of High Performance Fortran and MPI programs. The Annai tool environment has not been in development for several years.

The Performance Monitor and Analyser (PMA) is responsible for trace generation and performance analysis. Since it is designed for performance analysis, trace files are incomplete if processes fail during execution. PMA provides libraries to collect details of message transfers, execution of program routines and loops, memory utilisation, and other parallelisation and monitoring overheads.

3 GENERAL TRACE LIBRARY REQUIREMENTS

Despite extensive research into parallel program debugging techniques that employ trace data analysis, most MPI trace libraries target specific applications. Consequently there is a lack of general-purpose trace libraries.

In this section we identify the basic functionality that general-purpose trace libraries should provide. Although our research focuses on MPI, most concepts apply to any parallel programming environment. Whilst discussion concentrates on post-mortem trace collection, collecting trace data at run-time does not significantly affect the described functionality.

We note that some requirements are discussed only in broad terms, as they are difficult to specify, and that some requirements are not completely compatible with other requirements.

Ease of use

As most users expect software tools to provide results quickly, they are generally unwilling to invest time learning their use. Consequently, the effort required to obtain trace files should be minimal. In particular, trace libraries should require only minimal code changes to the application program, and should have a simple interface. The MPI profiling interface is known to be useful in this regard.

Minimisation of intrusion

All trace libraries must minimise program intrusion in order to minimise the probe effect, a particularly important issue for non-deterministic parallel programs (McDowell and Helmbold, 1989). Trace libraries cannot change program semantics, and should introduce only minimal

overhead. Instead, as much work as possible should be deferred until trace collection, such as determining globally unique communicator identifiers.

Usually frequent disk I/O is responsible for most intrusion. To alleviate this, trace events are often buffered in memory, and only written to temporary disk storage when necessary. Therefore, instead of writing in many small bursts, a smaller number of larger disk writes occur, which may allow the operating system to perform them more efficiently. Ideally the internal buffer size is large enough such that only one disk write is needed (at the conclusion of tracing), although for long running programs this is unlikely to be possible. However, whilst performing infrequent but large disk writes is generally preferred to the low but constant overhead introduced by frequent small writes, it is arguable that in some cases that the infrequent large writes may cause an apparent greater degree of intrusion.

Consistent trace generation

To be useful for correctness debugging, traces must be complete through to (abnormal or otherwise) program termination. Consequently, if trace events are buffered in memory, they must be written to disk prior to program completion regardless of the circumstances. This will usually involve catching a signal in the instance of abnormal program termination.

However, whilst catching signals and writing trace buffers upon detection of abnormal program termination is usually sufficient, it is insufficient when traced programs perform illegal memory accesses, in which case the trace buffers themselves could be corrupted. Therefore, the option of not buffering trace data in memory must be provided. Errors such as abnormal program termination should be indicated in the trace.

Configurable trace generation

It should be possible to disable trace data generation for individual MPI routines, for trace library features that are not always relevant, and for other trace library functionality that may incur a significant overhead of any form.

However, some trace data must always be generated, such as communicator creation and datatype creation information. No trace data should be absent if doing so compromises trace file integrity. For example, if communicator creation data is absent, determining which processes are expected to participate in collective operations is not possible.

Configurable trace collection

At times the user may be unsure what level of detail they require, so they may choose to trace everything possible, but wish not to analyse it all immediately. To support this, the trace collection utility should retrieve only the desired level of detail. Should this prove insufficient, the user should be able to re-collect the data at an increased level of detail without having to retrace the program.

For example, a user will often choose to retrieve only communication-related events. However, program errors may at times originate due to the incorrect use of non-communication routines. If a user is unsure what the cause is in advance, but the program is long running, it is beneficial to generate the trace only once, and it is convenient to restrict initial attention to those aspects of the program most likely in error, that is the communication-related aspects. However, the option is open to repeat trace collection in greater detail should doing so prove necessary.

Detailed trace data
Existing libraries do not trace all possible data. However, whilst it is easy to discard data, it is impossible to retrieve it. Thus, the option of tracing everything should remain available. For example, it should be possible to trace all entry and exit parameters to all MPI routines, report the addresses of pointer arguments, return codes, buffer addresses, and to some level of detail the contents of buffers.

Via the profiling interface it is a simple matter to disable the tracing of data that the user considers mundane, or when intrusion is an issue.

Tracing overhead
Each trace event should be accompanied by a value indicating the time consumed by the trace library in generating that trace data.

Clock synchronisation
Trace events should be accompanied by an associated entry and exit time. The times reported by the different processes should be synchronised as accurately as possible.

Source code reference
The associated source code line number and file name should accompany trace events where the compiler permits it.

Partially ordered logical clocks
Partially ordered logical clocks (Fidge, 1996) are useful with respect to correctness debugging, where, for example, they assist in determining actual event orderings (where clock skew adjustment is inadequate) and can be used to enumerate alternate possible event orderings.

Flexible trace format
It is very important that the trace file format be architecturally independent, flexible, and well defined. Moreover, since the requirements of all developers are too difficult to predict, the ability of the trace format to adapt to new requirements is particularly important.

Multiple trace files
Whilst most trace libraries store traces in a single file, this is unwieldy for large traces, particularly if only the events at the end of execution are of interest. Instead, it should be possible to represent the trace either as a single file, or across multiple (mostly autonomous) files. The trace should be split into multiple files based upon time, ideally according to a "happens before" relationship, although this may not always be feasible.

4 THE ROLE OF XML AND XML SCHEMA

XML is a meta-markup language used in many modern areas of computing to represent data both for storage and as a medium for communicating information between programs. XML provides a well defined markup syntax that specifies how and where items such as tags and elements appear. However, XML does not pre-specify what elements and tags appear, as these are specified by the program.

XML documents are custom-designed to suit the target application. It is the responsibility of the target application reading an XML document to apply semantic meaning to the data contained within.

XML is clearly specified, well structured, architecturally independent, and human readable. Moreover, XML is becoming increasingly well supported on a variety of architectures, with libraries to read and manipulate XML documents becoming increasingly common for an increasing range of programming languages.

XML is an entirely suitable syntax for parallel program traces. It is very general, capable of handling more demanding requirements than just trace data, it can be extended for unforeseeable future uses with relative ease, and third-party developers wishing to interpret XML-based trace files can do so using existing standard XML manipulation libraries.

XML satisfies the flexible trace format requirement that we make of trace libraries. In addition, XML documents can also contain comments, and processing instructions that target specific applications. Moreover, since XML is so flexible, it is also suited to dividing a trace into multiple files – for example, we could specify an index file such that it contains some basic trace information, with references to other trace file fragments.

The use of XML Schema can further specify the content and markup of XML documents. An XML Schema is used to specify the allowed content and structure of an XML file. When an XML file indicates it is using a schema, it indicates it conforms to the definitions within that schema. Schemas indicate what tags and elements, and their frequency, are allowed in correspondingly valid XML documents. Essentially, using schemas we can specify what events can occur in trace files, the data that must accompany the occurrence of each event, and the datatypes of the accompanying data. For example, a schema for an MPI trace file could specify that a `time` attribute is associated with a `FinalizeEntry` element, and that it is of the `double` datatype.

A schema-aware XML parser relieves applications from checking the content of XML files. In the context of using XML as a trace file format, applications using schema-aware parsers need not ensure elements are valid trace elements, and that traced parameters are of the correct type, as that is the schema-aware parser's responsibility. However, applications may wish to check that processes are, for example, still passing correct values to MPI routines.

Moreover, schemas are not limited to basic datatypes such as integers, doubles, and lists of integers. For example, with respect to MPI, we can indicate that traced parameters indicating message source ranks could appear as one of the following: a basic integer, the enumerated value `MPI_ANY_SOURCE`, or the enumerated value `MPI_PROC_NULL`. This has the effect of increasing the readability of trace files.

In addition, XML schemas are themselves human readable XML files. Their specification also makes them somewhat self-documenting; it is fairly clear what the content of conforming trace files can be.

XML Schema also facilitates extensibility. It is a simple matter for third-party developers to add definitions to an XML Schema, and doing so would not adversely affect earlier schema conforming XML files. Applications unaware of the new data can choose to discard it.

Although XML is not the most space efficient means of storing large amounts of data on disk, it is becoming one of the most accepted means of doing so. In particular, using XML as a trace file format would encourage third party developers to use it. The next section gives some example XML and XML Schema.

5 THE tracempi LIBRARY

To realise our trace library requirements, we are developing tracempi, an XML-based trace library for MPI programs, with the expectation to support tracing of all MPI 1.2 API routines in the C programming language, and later Fortran. The tracempi library consists of a trace generation component (where temporary binary trace data is written), and a trace collection component (where the binary data is read and output as XML).

Since tracempi uses the MPI profiling interface, to configure their program for a standard trace the user need only link it with tracempi without any code modification. This is because, as the MPI standard requests, tracing is at the default level after MPI_Init has been called. Greater control of tracempi is afforded via calls to MPI_Pcontrol using appropriate definitions obtained through the inclusion of a header file. MPI_Pcontrol is a routine specified by the MPI standard specifically for the purpose of providing a mechanism whereby the user can interface with profiling packages. In the absence of a profiling package, MPI_Pcontrol is a null routine.

The tracempi library has been successfully tested on a number of MPI programs.

5.1 Trace generation

After the program has been linked with the tracempi library, it is executed in the normal manner. During execution, trace data is written to temporary disk storage at each processing node. This data is written as raw binary data to minimise file size and processing overhead, and thus minimise trace generation overhead.

The temporary storage of binary data is not expected to be analysed by third-party tools or the user, as it is only intended for use by the tracempi library.

5.2 Trace collection

After either successful or unsuccessful program termination, the user must run the trace collection program collect, itself an MPI program. During execution, collect reads each of the temporary binary trace files, amalgamates and sorts the trace events, and generates the target XML trace file.

By collecting the trace data subsequent to program execution, intrusion is minimised. The alternative, in which the trace is collected in user space prior to program termination, is subject to failure if the MPI subsystem fails due to program error.

5.3 Trace format

We have defined a preliminary XML Schema against which tracempi traces are valid.

For every MPI 1.2 API function there is a corresponding entry and exit event in the schema, as MPI routines are traced prior to invoking the requested MPI functionality, and after the actual MPI routine has completed. The schema also defines supporting type definitions, and supporting trace data events.

For example, consider the schema definition for the MPI_Gather entry event:

```
<complexType name="GatherEntry">
    <sequence>
        <element name="root" type="int"/>
        <element name="sendcnt" type="int"/>
        <element name="sendtype" type="trc:datatype"/>
```

```
            <sequence minOccurs="0">
                <element name="recvcnt" type="int"/>
                <element name="recvtype" type="trc:datatype"/>
            </sequence>
            <element name="comm" type="trc:communicator"/>

            <sequence minOccurs="0">
                <element name="sendbuf_A" type="trc:address"/>
                <element name="recvbuf_A" type="trc:address"
                         minOccurs="0"/>
            </sequence>
        </sequence>

        <attributeGroup ref="trc:EntryEvent"/>
    </complexType>
```

In English, this definition indicates that the `GatherEntry` XML element consists of an ordered sequence of elements including:

- the rank of the receiving process, `root` (an integer),
- the number of items in the send buffer, `sendcnt` (an integer),
- the datatype of the send buffer items, `sendtype` (a custom defined datatype `datatype`),
- an optional pair of elements (that appear together or not at all), the number of items for any single receive (`recvcnt`, an integer), and the datatype of receive buffer items (`recvtype`, a custom defined datatype `datatype`),
- the communicator, `comm` (a custom defined datatype `communicator`),
- the optional element `sendbuf_A` (a custom type named `address`), which may optionally be accompanied by the element `recvbuf_A` (also of type `address`), and
- the set of attributes as custom-defined by `EntryEvent`.

The elements of `GatherEntry` directly correspond to the information traced upon entry to the `MPI_Gather` routine. We note that the elements `recvcnt` and `recvtype` are optional since they are only significant at the root process. Also, `sendbuf_A` and `recvbuf_A` are collectively optional (in that they occur within the same optional sequence) since they represent memory addresses. Additionally, `recvbuf_A` is singularly optional as it is significant only at the root process, so if it appears, it must appear alongside `sendbuf_A`, since `sendbuf_A` is not optional within the sequence in which it appears.

Consider the above schema definition, and an MPI program written in C making the call `MPI_Gather(&sendbuf, 1, MPI_INT, &recvbuf, 1, MPI_INT, 0, MPI_COMM_WORLD)`, 2.002455 seconds after `MPI_Init` concluded, and in process rank 0. The `tracempi` library would "intercept" this call, where interception is possible by linking the user code against the `tracempi` library, which causes the user code to directly call the trace library routines, which in turn invoke the actual MPI routines – this is a standard manner of tracing MPI programs. During interception the trace library would store a corresponding entry and exit event in temporary storage. Later, the MPI program `collect` would collect these events and produce the following XML for the entry event (the corresponding exit event is not shown):

```
<GatherEntry rank="0" time="2.002455">
    <root>0</root>
    <sendcnt>1</sendcnt>
    <sendtype>INT</sendtype>
    <recvcnt>1</recvcnt>
    <recvtype>INT</recvtype>
    <comm>COMM_WORLD</comm>
    <sendbuf_A>3221222972</sendbuf_A>
    <recvbuf_A>134692432</recvbuf_A>
</GatherEntry>
```

It is worth noting that whilst it is neither easier nor harder to use an XML schema to specify trace events than it would be to explicitly assume a trace format (as has been done in the past), the schema makes the trace format clearer. Moreover, it eases the development of a trace parser. However, defining a schema by hand for over 100 MPI routines is relatively tedious, which possibly makes employing a higher-level specification language to automatically generate schema definitions worthwhile.

5.5 Future extensions to `tracempi`

Beyond the requirements specified in Section 3, we also expect to make a number of enhancements to `tracempi`. The first enhancement involves automatically compressing traces on the fly in order to occupy less space, whilst not adversely affecting their accessibility. Commonly available stream-based compression and decompression libraries support this approach. In our experiments, trace files produced by the `tracempi` library can often be compressed to as little as 15% of their original size. However, compressing and decompressing trace files takes roughly 40% longer when compared to using plain trace files.

6 CONCLUSION

Trace files are an important asset to the parallel programmer since they are applicable to both correctness and performance debugging, and can be repeatedly analysed. However, for any form of formatted data, including trace data, to ever gain wide acceptance, it must be in a form that is architecturally independent, well defined, and extensible. With the advent of XML and its increasing acceptance as a standard form of representing data, it is the logical choice to use as the format for parallel program trace data. In this paper we have discussed what is desirable of a general-purpose trace library, and we have shown the advantages of using XML and XML Schema. As part of demonstrating the feasibility of the indicated requirements, we have implemented a trace library `tracempi`, which uses XML as its trace file format.

At present the `tracempi` library is capable of tracing most MPI routines. Moreover, at the time of publishing the `tracempi` library satisfied various of the trace library requirements described herein, including the ease of use requirement, the minimisation of intrusion requirement, the consistent trace generation requirement, the configurable trace generation requirement, the clock synchronisation requirement, and the flexible trace format requirement. Most other requirements have been partially satisfied.

We are continuing to work on `tracempi` in order to provide added functionality and to fulfil the remaining specified requirements. In addition, we are developing a topology-based debugger

DEPICT (Debugger of Parallel but Inconsistent Communication Traces) (Huband and McDonald, 2001) that will use and test the capabilities of `tracempi`.

7 REFERENCES

AYDT, R.A. (1992): The Pablo self-defining data format. Technical Report. Department of Computer Science, University of Illinois.

BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C.M. and MALER, E. (eds) (2000): Extensible markup language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000. Available at: `http://www.w3.org/TR/2000/REC-xml-20001006.`

DE ROSE, L. and REED, D.A. (1999): SvPablo: A multi-language architecture-independent performance analysis system. Proceedings of the 1999 International Conference on Parallel Processing, Fukushima, Japan, 311-318, IEEE Computer Society Press.

FALLSIDE, D.C. (ed) (2001): XML Schema Part 0: Primer, W3C Proposed Recommendation, 30 March 2001. Available at: `http://www.w3.org/TR/2001/PR-xmlschema-0-20010330/.`

FIDGE, C.J. (1996): Fundamentals of distributed system observation. IEEE Software 13(6):77-83.

GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R. and SUNDERAM, V. (1994): PVM 3 user's guide and reference manual. Available at: `http://www.netlib.org/pvm3/.`

GROPP, W., LUSK, E., DOSS, N. and SKJELLUM, A. (1996): A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22(6):789-828.

HEATH, M.T. and ETHERIDGE, J.A. (1991): Visualizing the performance of parallel programs. IEEE Software 8(5):29-39.

HERRARTE, V. and LUSK, E. (1991): Studying parallel program behavior with Upshot. Technical Report ANL-91/15. Mathematics and Computer Science Division, Argonne National Laboratory.

HUBAND, S. and MCDONALD, C. (2001): A preliminary topological debugger for MPI programs. Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid – CCGrid2001, Brisbane, Australia, 422-429.

KOHL, J.A. and GEIST, G.A. (1996): The PVM 3.4 tracing facility and XPVM 1.1. Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences, Wailia, USA, 290-299, EL REWINI, H. and SHRIVER, B.D. (eds), IEEE Computer Society Press.

KRANZLMÜLLER, D. (2000): Event graph analysis for debugging massively parallel programs. Ph.D. thesis. Johannes Kepler University Linz, Austria.

KRANZLMÜLLER, D., GRABNER, S., and VOLKERT, J. (1997): Debugging with the MAD environment. Parallel Computing 23(1-2):199-217.

MCDOWELL, C.E. and HELMBOLD, D.P. (1989): Debugging concurrent programs. ACM Computing Surveys 21(4):593-622.

MESSAGE PASSING INTERFACE FORUM (1995): MPI: A message-passing interface standard (version 1.1). Technical Report. University of Tennessee. Available at: `http://www.mpi-forum.org.`

MESSAGE PASSING INTERFACE FORUM (1997): MPI-2: Extensions to the message passing interface. Technical Report. University of Tennessee. Available at: http://www.mpi-forum.org.

NAGEL, W.E., ARNOLD, A., WEBER, M., HOPPE, H.Ch. and SOLCHENBACH, K. (1996): VAMPIR: Visualisation and analysis of MPI resources. Supercomputer 12(1):69-80.

REED, D.A., AYDT, R.A., NOE, R.J., ROTH, P.C., SHIELDS, K.A., SCHWARTZ, B.W. and TAVERA, L.F. (1994): Scalable performance analysis: the Pablo performance analysis environment. Proceedings of the Scalable Parallel Libraries Conference, 104-113, IEEE Computer Society Press.

RONSSE, M., DE BOSSCHERE, K. and CHASSIN DE KERGOMMEAUX, J. (2000): Execution replay and debugging. Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG 2000), Munich, Germany, 5-18, DUCASSÉ, M. (ed).

WORLEY, P.H. (1992): A new PICL trace file format. Technical Report ORNL/TM-12125. Oak Ridge National Laboratory, Oak Ridge.

WYLIE, B.J.N. and ENDO, A. (1996): The Annai/PMA performance monitor and analyser. MASCOTS '96. Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, San Jose, USA, 186-191, IEEE Computer Society Press.

YAN, J., SARUKKAI, S. and MEHRA, P. (1995): Performance measurement, visualisation and modeling of parallel and distributed programs using the AIMS toolkit. Software – Practice and Experience 25(4):429-461.

ZAKI, O., LUSK, E., GROPP, W. and SWIDER, D. (1999): Toward scalable performance visualisation with Jumpshot. High Performance Computing Applications 13(2):277-288.

BIOGRAPHICAL NOTES

Simon Huband is studying for his PhD in the the Department of Computer Science & Software Engineering at The University of Western Australia, where he is an active member of the Programming, Languages and Systems group. His primary field of research is parallel program debugging, in particular the debugging of MPI and other message-passing programs. Simon also has a somewhat less active interest in NP-hard problems.

Chris McDonald holds a BSc(Hons) and PhD in Computer Science from The University of Western Australia. His graduate studies addressed the formal specification of command language interpreters, in particular the application of functional programming techniques to command interpreter design and use. His current research interests include parallel programming environments, network simulation, the Linda parallel programming model, programming language implementation, and the LINUX operating system.

Practical Experience Using a Computational Model for the Design of Heterogeneous Distributed Software

Tiffani Williams and Rebecca Parsons

University of Central Florida
School of EECS
Orlando, FL 32816-2362 USA
williams,rebecca@cs.ucf.edu

Heterogeneous cluster environments are becoming an increasing popular platform for executing parallel applications. Efficient heterogeneous programs must account for the differences inherent in such an environment. We propose the HBSP^1 model of computation as a framework for developing applications for heterogeneous clusters of workstations. The utility of the model is demonstrated through the design and analysis of the scatter and one-to-all broadcast algorithms. Extensive experimentation illustrates the benefits of using the model for heterogeneous program development. By hiding the non-uniformity of the underlying system, the HBSP^1 model provides a framework that embraces the heterogeneity of the underlying system.

Keywords: heterogeneous distributed computing, BSP, cluster computing, collective communication, performance evaluation

1. INTRODUCTION

Heterogeneous cluster environments are becoming an increasing popular platform for executing parallel applications. Such environments consist of a collection of machines with myriad differences such as varying computational power and incompatible data formats. Performance gains in heterogeneous environments result from effectively exploiting the speeds of the underlying components. Executing standard (homogeneous) distributed applications on heterogeneous platforms leads to low-end systems becoming a bottleneck, which reduces overall system performance. Thus, a new approach is necessary for the design of efficient heterogeneous distributed applications.

The k-Heterogeneous Bulk Synchronous Parallel model (HBSP^k) is the model that we propose for the development of general-purpose heterogeneous applications (Williams, 2000). It is an extension of the BSP model of parallel computation (Valiant, 1990). The superscript k refers to the number of network layers present in the heterogeneous environment. For example, $k = 0$ refers to a single-processor computer, $k = 1$ models a network of workstations, and $k = 2$ denotes a cluster of clusters environment. Here, we focus on the development of programs for a heterogeneous cluster of workstations, which consist of a single communication network. Thus, a specific instantiation of the generalised HBSP^k model, known as the HBSP^1 model, is used to develop heterogeneous parallel algorithms for workstation clusters.

Copyright© 2001, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 16 April, 2001, revised 16 August, 2001
Associate Editor: Michael Oudshoorn

Collective communication algorithms are used frequently as building blocks in a variety of distributed algorithms. Proper implementation of these operations is vital to the efficient execution of the distributed algorithms that use them. Collective operations designed for homogeneous distributed systems are not adequate for heterogeneous environments. As a result, we present two collective communication algorithms – scatter and one-to-all broadcast – for a heterogeneous cluster of workstations. Our HBSP^1 algorithms are based on BSP communication routines (Hill, 1997; Juurlink, 1996). Our design strategy, which is guided by the HBSP^1 model, for these algorithms is two-fold. Faster workstations should be involved more in the computation than slower machines. Secondly, faster nodes should receive more data items than slower nodes. The HBSP^1 model predicts that increased performance will result if these guidelines are taken into consideration when designing heterogeneous applications.

We perform several experiments to validate the predictions of the model. Our experimental testbed consists of a non-dedicated, heterogeneous cluster of workstations. Experimental results demonstrate that our collective algorithms have increased performance on heterogeneous platforms. Moreover, the model accurately predicts the performance trends of the communication algorithms. Improved performance is not a result of programmers having to account for myriad differences in a heterogeneous environment. By hiding the non-uniformity of the underlying system from the application developer, the HBSP^1 model offers a framework that encourages the design of software for heterogeneous clusters in an architecture-independent manner.

The rest of this paper is organised as follows. Section 2 gives a brief overview of related work. The HBSP^1 model is described in Section 3. Heterogeneous algorithm design is discussed in Section 4. Sections 5 and 6 present our experimental approach and the experimental results, respectively. Conclusions are given in Section 7.

2. RELATED WORK

The Bulk Synchronous Parallel (BSP) (Valiant, 1990) model provides the foundation for the HBSP^1 model. The BSP model provides guidance on designing applications for good performance on homogeneous parallel machines. Support for BSP includes theoretical results, empirical results, and experimental parameterisation of BSP programs (Gerbessiotis, 1994; Goudreau, 1999).

Two models that address heterogeneous clusters of workstations are the Heterogeneous Coarse-Grained Multicomputer (HCGM) model (Morin, 1998) and the Heterogeneous Bulk Synchronous Parallel (HBSP^1) (Williams and Parsons, 2000), which is synonymous with HBSP^1. Both of these models take into account varying processor speeds to develop parallel algorithms for heterogeneous systems. The main difference between the two models is that HCGM is not intended to be an accurate predictor of execution times whereas HBSP^1 attempts to provide the developer with predictable algorithmic performance.

Additional research has studied the performance of collective algorithms for heterogeneous workstation clusters. The ECO package (Lowekamp and Beguelin, 1996), built on top of PVM (Sunderam, 1990), automatically analyses characteristics of heterogeneous networks to develop optimised communication patterns. Bhat, Raghavendra and Prasanna (1999) extend the FNF algorithm (Banikazemi, Moorthy and Panda, 1998) and propose several new heuristics for collective operations. Their heuristics consider the effect communication links with different latencies have on a system. Banikazemi, Sampathkumar, Prabhu, Panda and Sadayappan (1999) present a model for point-to-point communications in heterogeneous networks of workstations and use it to study the effect of heterogeneity on the performance of collective operations.

3. THE HBSP^1 MODEL

The HBSP^1 model is a synchronous model of computation that provides a framework for the design of software for a heterogeneous cluster of workstations. Its objective is to describe heterogeneous platforms in simple but realistic terms. HBSP^1 borrows heavily from the BSP model of parallel computation. The essence of the HBSP^1 approach is the notion of the superstep, and the idea that the input/output (i.e., sends and receives) associated with a superstep is performed as a global operation. An HBSP^1 computation consists of a sequence of supersteps. During a superstep, each processor performs asynchronously some combination of local computation, message transmissions, and message arrivals. A message sent in one superstep is guaranteed to be available to the destination processor at the beginning of the next superstep. Each superstep is followed by a global synchronisation of all the processors.

3.1 Model parameters

Formally, the HBSP^1 model can be described by the following parameters:

- p, the number of processors or workstations labeled P_0,...., P_{p-1};
- g, a bandwidth indicator that reflects the speed with which the *fastest* machine can inject messages into the communications network;
- r_j, the speed relative to the *fastest* processor for P_j to inject a packet into the network;
- L, the overhead to perform a barrier synchronisation of the p processors; and
- c_j, the fraction of the problem size that P_j receives.

For notational convenience, the indices f and s identify the fastest and slowest nodes, respectively. We assume that the r_f value of the fastest machine is normalised to 1. If $r_j = t$, then P_j communicates t times slower than the fastest workstation. More specifically, P_j requires a cost of $g \cdot r_j$ to inject a message into the network.

The c_j parameter adds a load-balancing feature into the model. It's value is in the range from 0 to 1. Typically in a homogeneous environment $c_j = \frac{1}{p}$ for all j. The parameter c_j attempts to provide P_j with a problem size that is proportional to its computational and communication abilities. Intuitively, c_j is inversely proportional to r_j.

3.2 Cost model

Since HBSP^1 defines a specific programming style, the formal parameters of the model allow for the cost analysis of HBSP^1 programs. Again, the basic notion of an HBSP^1 computation is the superstep, which consists of local computation, communication, and synchronisation. Let w represent the largest amount of local computation performed by a workstation. Let h_j be the largest number of messages sent or received by processor j. The size of the *heterogeneous h-relation* is $h = \max\{r_j \cdot h_j\}$ requiring a communication cost of gh. Thus, the cost of a superstep is

$$w + gh + L. \tag{1}$$

The overall cost of the program is the sum of the superstep times.

The above cost model demonstrates what factors are important when designing HBSP^1 applications. To minimise execution time, the programmer must attempt to (i) balance the local computation in each superstep, (ii) balance the communication between the machines, and (iii) minimise the number of supersteps. Balancing these objectives is a nontrivial task. Nevertheless, HBSP^1 provides assistance with making the tradeoffs necessary for the design of efficient heterogeneous programs.

4. HETEROGENEOUS ALGORITHM DESIGN

The HBSP^1 model provides parameters that allow application developers to exploit the heterogeneity of the underlying system. The model promotes a two-fold strategy for designing heterogeneous collective operations. First, faster machines should be involved in the computation more often than their slower counterparts. Collective operations use specific nodes to collect or distribute data to the other nodes in the system. For faster algorithmic performance, these nodes should be the fastest machines in the system. Secondly, faster machines should receive more data items than slower machines. This principle encourages the use of *balanced* workloads, where machines receive problem sizes relative to their communication and computational abilities. Partitioning the workload so that nodes receive an equal number of elements works quite well for homogeneous distributed environments. However, this strategy encourages *unbalanced* workloads in heterogeneous environments since faster machines typically sit idle waiting for slower nodes to finish a computation.

Throughout the rest of this section, let n represent the total number of items of interest. Balanced workloads assume P_j possesses $c_j n$ elements.

4.1 Scatter

The scatter operation uses a single root node to distribute a unique message to each of the other nodes. Here, each processor j will receive $c_j n$ unique data items from P_f. In the homogeneous version, each node receives $\frac{n}{p}$ elements. The HBSP^1 scatter algorithm requires a single superstep. Therefore, the size of the single, heterogeneous h-relation is $\max\{r_j \cdot c_j n, r_f \cdot n\}$. Each processor's r_j value is relative to the fastest processor. Hence, $r_f = 1$ and $r_j \geq r_f$. Recall that c_j is inversely proportional to the speed of P_j. Consequently, $r_j c_j < 1$. Thus, the HBSP^1 scatter cost is $gn + L$.

The above cost of the scatter operation is efficient since the fastest processor is performing most of the work. If $r_j c_j > 1$, P_j has a problem size that is too large. Its communication time will dominate the cost of the scatter operation. Whenever possible, the fastest processor should handle the most data items. Our analysis demonstrates the importance of balanced workloads. Thus, the HBSP^k model rewards programs with balanced design.

4.2 One-to-all broadcast

In the one-to-all broadcast, only the source processor has the data that needs to be broadcast. At the termination of the procedure, each node has a copy of the data. The HBSP^1 broadcasts executes similarly to the two-phase BSP algorithm (Hill *et al*, 1997). During the first phase, the root node distributes $\frac{n}{p}$ items to each processor. Afterwards, processor j is responsible for sending its share of the data to its peers.

During the first phase of the algorithm, P_j receives $\frac{n}{p}$ items from P_f. This phase requires a heterogeneous h-relation of size $\max\{r_f \cdot n, r_j \cdot \frac{n}{p}\}$. In a typical environment, it is reasonable to assume that p ranges from the tens to the hundreds. It is quite unlikely that a machine would communicate p times slower than the fastest machine. If this is the case, it may be more appropriate not to include that machine in the computation. As a result, the communication time of the first phase reduces to gn. During the second phase, each processor must receive the same number of items. Thus, the slowest processor will cause a bottleneck. Let r_s represent the communication time of the slowest node. This results in a communication time of $gr_s n$. Actually, P_s will receive $n - \frac{n}{p}$ elements. We use n to simplify the notation. Thus, the complexity of a two-phase broadcast on an HBSP^1 machine is $gn(1 + r_s) + 2L$.

As a point of comparison, the one-phase broadcast (P_f sends n items to each of its children) costs $gnp + L$. For reasonable values of r_s, the two-phase approach is the better overall performer. An interesting conclusion concerning the broadcast operation is that it effectively cannot exploit heterogeneity. Since the slowest processor must receive n items, its cost will dictate the complexity of the algorithm. Moreover, partitioning the problem size based on the c_j parameter is ineffective. Although wall clock performance may improve, theoretically, the resulting speedup is negligible.

5. EXPERIMENTAL APPROACH

Using HBSP[1] as a guide, we have designed and analysed the scatter and one-to-all broadcast operations. According to the model, the algorithms should demonstrate good performance on a heterogeneous cluster of workstations. We are now ready to investigate the behaviour of these algorithms on an actual heterogeneous platform. In this section, we describe our experimental methodology and Section 6 provides the experimental results.

5.1 HBSP*lib*

Our collective communication algorithms are implemented using the HBSP Programming Library (HBSP*lib*). Table 1 lists the functions that constitute the HBSP*lib* interface. The design of HBSP*lib* incorporates many of the functions contained in BSP*lib* (Hill, McColl, Stefanescu, Goudreau, Lang, Rao, Suel, Tsantilas and Bisseling, 1998). HBSP*lib* is written on top of PVM (Sunderam, 1990), a software package that allows a heterogeneous network of computers to appear as a single, concurrent, computational resource. The computers compose a *virtual machine* and communicate by sending messages to each other. We use PVM's pvm_send() function for asynchronous communication to directly send messages between heterogeneous processors. To receive a message, we take advantage of the PVM function `pvm_recv()`. The `pvm_barrier()` primitive provided

Function	*Semantics*
hbsp_begin	Starts the program with the number of processors requested.
hbsp_end	Called by all processors at the end of the program.
hbsp_abort	One process halts the entire HBSP computation.
hbsp_pid	Returns the processor id in the range of 0 to one less than the number of processors.
hbsp_time	Returns the time (in seconds) since `hbsp_begin` was called. The timers on each of the processors are not synchronized.
hbsp_nprocs	Returns the number of processors.
hbsp_sync	The barrier synchronization function call. After the call, all outstanding requests are satisfied.
hbsp_send	Sends a message to a designated processor.
hbsp_get_tag	Returns the tag of the first message in the system queue.
hbsp_qsize	Returns the number of messages in the system queue.
hbsp_move	Retrieves the first message from the processor's receive buffer
hbsp_get_rank	Returns the identity of the processor with the requested rank.
hbsp_get_speed	Returns the speed of the processor of interest.
hbsp_cluster_speed	Returns the total speed of the heterogeneous cluster.

Table 1: The functions that constitute HBSP*lib* interface

by PVM assisted with the development of `hbsp_sync()`. However, our implementation of global synchronisation is somewhat complex since we needed to guarantee that all messages arrived at their destination before the beginning of the next superstep.

HBSP*lib* incorporates functions that allow the programmer to take advantage of the heterogeneity of the underlying system. Under HBSP[1], faster machines should perform the most work. The primitive `hbsp_get_rank(1)` returns the identity of the fastest processor. `hbsp_get_rank(p)` returns the slowest machine's identity, where p is the number of processors. HBSP*lib* also includes functions to help the programmer distribute the workload based on a machine's ability. The HBSP*lib* primitive `hbsp_get_speed(j)` provides the speed of processor j. `hbsp_cluster_speed` returns the speed of the entire cluster. When combined together, these two functions allow for finding the value of processor j's c_j parameter. We discuss the calculation of c_j in more detail in Section 5.4.

Figure 1 shows the implementation of the scatter algorithm using HBSP*lib*. The algorithm requires four parameters: `sendbuf`, which contains the data items the root node sends to the other processors; `sendcounts`, which is an array that tells the root node the number of elements that each processor should receive (i.e., the root will send `sendcounts[j]` elements to P_j); `recvbuf`, where the nodes store the items received from the root node; and `root`, the identity of the source node. The algorithm first requires the root node to send the data to all of the other processors. In order to send the data, `hbsp_send` requires the destination, a tag to identify the message (if relevant), the beginning address of the data buffer, and the size of the data to be communicated. In the second superstep, each processor puts the data sent to it from the root into its `recvbuf`.

Host	CPU type	CPU speed (MHz)	Memory (MB)	Data cache (KB)
aditi‡	UltraSPARC II	360	256	16
chromus	microSPARC II	85	64	8
dcn_sgi1	MIPS R5000	180	128	32
dcn_sgi3	MIPS R5000	180	128	32
gradsun1	TurboSPARC	170	64	16
gradsun3	TurboSPARC	170	64	16
gromit	UltraSPARC IIi	333	128	16
sgi1	MIPS R5000	180	96	32
sgi3	MIPS R5000	180	96	32
sgi7	MIPS R5000	200	64	32

‡ A 2 processor system, where each number is for a single CPU.

Table 2: Specification of the nodes in our heterogeneous cluster.

5.2 Experimental platform

Our experimental testbed consists of a non-dedicated, heterogeneous cluster of SUN and SGI workstations at the University of Central Florida. Table 2 lists the specifications of each machine. Our platform is quite heterogeneous. CPU speeds range from 85MHz to 360MHz and memory sizes vary between 64MB to 256MB. Each node is connected by a 100Mbit/s Ethernet connection.

5.3 Machine ranking

The ranking of the heterogeneous nodes is determined by the BYTEmark benchmark (`http://www.byte.com/bmark/bmark.htm`), which consists of a variety of different tests that extensively exercise a machine's capabilities. A sampling of programs in the benchmark

```
void bsp_scatter(int *sendbuf, int *sendcounts, int *recvbuf, int root)
{
   int bytes, i, j, offset, size, temp;

   /* root sends data to the processors */
   if (hbsp_pid() == root) {
      offset = 0;
      for (i = 0; i < p; i++) {
         if (root != i)
             hbsp_send(i, NULL, sendbuf+offset, sendcounts[i] *sizeof(int));
         else
             temp = offset;
         offset += sendcounts[i];
       }

       /* root copies its data into recvbuf */
       size = sendcounts[root];
       for (i = 0; i < size; i++)
          recvbuf[i] = sendbuf[i+temp];
   }
   hbsp_sync();

   /* processors receive data from root */
   if (hbsp_pid() != root) {
      hbsp_get_tag (&bytes, NULL);
      bsp_move(recvbuf, bytes);
   }
}
```

Figure 1: The scatter algorithm written using HBSP*lib*.

suite include numeric and string sorting, an IDEAL encryption algorithm, Huffman compression, a floating-point package, a back-propagation network simulator, and a LU Decomposition solver.

After running all of the tests, BYTEmark produces two overall figures, an Integer and a Floating-point index. The Integer index is the geometric mean of those tests that involve only integer processing. The remaining tests comprise the Floating-point index. Since the benchmark is a few years old, the index score calculation is based on the performance of a 90MhZ Pentium. If a machine has an index score of 2.0, it is twice as fast a 90MhZ Pentium computer.

Table 3 presents the Integer and Floating-point index scores for each machine in the heterogeneous cluster. Since we consider integer data only, the Integer index scores were used to rank the processors. According to the results, `chromus` is the slowest node. `gromit` is the fastest machine in the cluster. This result is surprising considering `aditi` appears faster on paper. Interestingly, `aditi` narrowly edges out `gromit` in every test, except string sort, where `gromit` outperforms `aditi` with a score of 7.63 to 2.40. Since BYTEmark uses only a single execution thread, it cannot take advantage of `aditi`'s additional processor. This does not present a problem for our experiments since our HBSP*lib* implementation does not use threads. We ran our experiments with both `aditi` and `gromit` as the fastest processor. There was no major difference in the execution times. Therefore, we consider `gromit` to be the fastest processor in the cluster.

Machine	Integer Index	Floating-point Index
aditi	4.45	3.77
chromus	0.75	0.59
dcn_sgi1	2.80	3.73
dcn_sgi3	2.79	3.67
gradsun1	1.80	1.41
gradsun3	1.81	1.42
gromit	4.89	3.33
sgi1	2.81	3.60
sgi3	2.77	3.30
sgi7	3.13	4.11

Table 3: BYTEmark benchmark scores.

5.4 Parameter estimation

In order to compare the actual and predicted (theoretical) execution times of the algorithms, we must determine the values of the HBSP^1 parameters on an actual heterogeneous platform. Below, we describe our method for finding the values of the c_j, L, r_j, and g parameters of the HBSP^1 model. It is important to note that these are architecture-dependent parameters. If we were to change the underlying platform, we would need to recalculate the parameter values for that environment.

Unlike a homogeneous environment, the ordering of the processors can have a dramatic effect on the performance results. To ensure consistent results, we apply the same processor ordering for each experiment. Table 4 shows the ordering. When $p = 2$, the experiments utilise `gromit` and `chromus`. The speed of this configuration is 5.64, which is the sum of each machine's Integer index score. Each machine's c_j value is based on its Integer index score and the cluster speed. When $p = 2$, `gromit`'s c_j value is $\frac{4.89}{5.64}$ (or 0.867). The c_j value of `chromus` is 0.133. Therefore, `gromit` receives 86.7% of the data elements and `chromus` acquires the remaining 13.3%. When $p = 4$, the cluster speed is 12.89. The workstations that comprise the cluster are `gromit`, `chromus`, `aditi`, and `dcn_sgi1`, which receive 37.9%, 5.8%, 34.5%, and 21.7% of the input, respectively.

Table 4 also presents the synchronising costs of the clusters comprised of 2, 4, 6, 8, and 10 workstations. For example, synchronising two processors (i.e, `gromit` and `chromus`) requires 9,000 μs. The value of L corresponds to the time for an empty superstep (i.e., no computation or

p	Machine	Speed	$L(\mu s)$
2	gromit, chromus	5.64	9,000
4	aditi, dcn_sgi1	12.89	15,000
6	dcn_sgi3, gradsun1	17.48	23,000
8	gradsun3, sgi1	22.10	30,000
10	sgi3, sgi7	28.00	37,000

Table 4: Cluster speed and synchronisation costs.

machine	r_j
aditi	1.03
chromus	4.08
dcn_sgi1	2.12
dcn_sgi3	1.95
gradsun1	2.00
gradsun3	2.46
gromit	1.00
sgi1	1.68
sgi3	1.20
sgi7	1.16

Table 5: r_j values

communication). When $p = 4$, 15,000 μs are needed in order to synchronise the processors. Several factors contribute to the high synchronisation costs. Since the cluster is non-dedicated, many other nodes share the network link, which effectively degrades communication performance. Secondly, our implementation of barrier synchronisation is not necessarily efficient. Despite the high L values, our collective algorithms outperformed their PVM counterparts. Additional work will focus on the development of a more efficient barrier synchronisation primitive.

Table 5 shows the r_j values achieved on our heterogeneous cluster. To obtain these values, we measure the time needed for each machine to inject a sufficiently large packet into the network. `gromit` performed the best with a score of 0.196 $\frac{\mu s}{byte}$, which is the value of g. Processor j's r_j value is relative to this score.

6. EXPERIMENTAL RESULTS

The input data for each experiment consists of 100KBytes to 1000KBytes of uniformly distributed integers. The problem size, n, refers to the largest number of integers possessed by the root. Experimental results are given in terms of an improvement factor. Let T_A and T_B represent the execution time of algorithm A and algorithm B, respectively. The improvement factor of using algorithm B over algorithm A is $\frac{T_A}{T_B}$.

The HBSP^1 model encourages the use of fast processors and balanced workloads. According to the model, applications that embody both of these principles will result in good performance. We designed two types of experiments to validate the predictions of the model. The first experiment tests whether processor speed has an impact on algorithmic performance. Let T_s represent the execution time of a collective routine assuming the root node is the slowest processor, P_s. T_f denotes the algorithmic cost of using P_f as the root. For these experiments, each processor has an equal number of data items since our objective is to monitor the performance of slow versus fast root nodes. Hence, $c_j = \frac{1}{p}$.

Our second experiment studies the benefit of using the fastest processor as the root and balanced workloads. Let T_u be the execution time when the workload is unbalanced. Note that $T_u = T_f$. Each processor j's c_j value is $\frac{1}{p}$. T_b denotes the execution time when the workload is balanced. Here, c_j is computed as described in the Section 5.4.

We also investigate the accuracy of the HBSP^1 cost function in predicting execution times. Similarly to BSP, we consider HBSP^1 to model only communication and synchronisation (Goudreau *et al*, 1999). I/O and local computation are not modelled. As a result, none of our experiments include I/O. Furthermore, the work component (w) of our algorithms is neglible. As a result, the cost model that we use to predict the cost of a superstep is $gh + L$.

The remainder of this section provides experimental results for the scatter and one-to-all broadcast operations. Complete experimental results can be found in Williams (2000). Each data point is the average of 10 runs. For each of the experiments, the logic of the algorithms is not changed. Instead, the modifications occur in either root node selection (fast versus slow root node) or problem size distribution (balanced versus unbalanced workloads).

6.1 Scatter

Figure 2 (a) plots the increase in performance if the root node is the fastest processor. The improvement factor is steady as the problem size increases. The best improvement occurs when $p = 6$ and $n = 500$KB. When $p = 2$, $\frac{T_s}{T_f} < 1$. Figure 2 (b) compares the performance of unbalanced and balanced workloads. The results indicate that there is a benefit to distributing the problem size

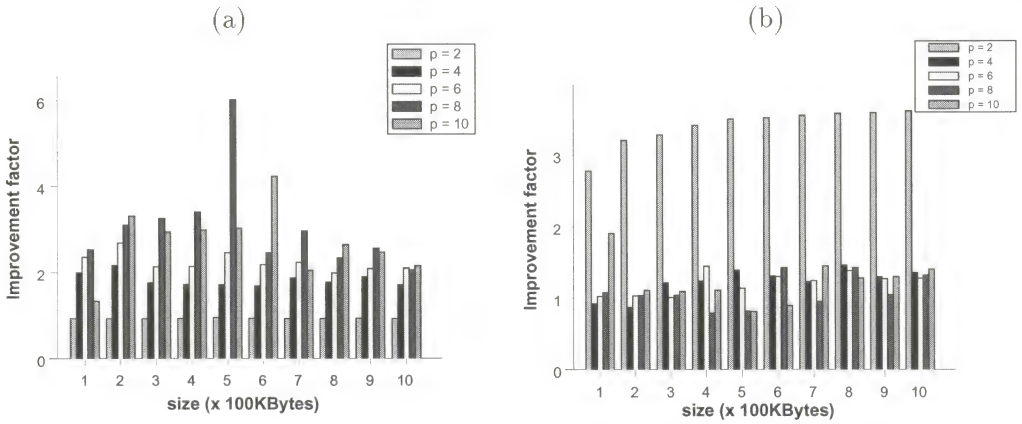


Figure 2: Scatter actual performance. The improvement factor is determined by (a)$\frac{T_s}{T_f}$ and (b)$\frac{T_u}{T_b}$. The problem size ranges from 100KB to 1000KB of integers. Each data point represents the predicted performance on a cluster comprised of 2, 4, 6, 8 and 10 heterogeneous processors.

based upon a processor's computational abilities. Here, $p = 2$ had the best performance with a maximum improvement of 3.62. Figure 3 shows predicted performance for the scatter operation.

For both experiments, the results at $p = 2$ are interesting. First, Figure 2 (a) shows that it is better for the root node to be the slowest workstation. This seems counterintuitive. In our implementation of scatter (as well as the other collective operations), a processor does not send data to itself. When P_s is the root, P_f receives $\frac{n}{p}$ items from it. Similarly, if the fastest processor is the root, P_s receives $\frac{n}{p}$ elements from P_f. $T_s < T_f$ implies that it is more beneficial to have P_f waiting on data from P_s. Clearly, the root node should be P_f as the number of processors increase.

Secondly, at $p = 2$, balanced workloads contribute to increased performance. T_u is the execution time of P_s receiving $\frac{n}{p}$ data elements from the fastest processor. T_b is the cost of P_s receiving $c_s n$ integers from P_f, where c_s is calculated as described in Section 5.4. Note that $c_s n < \frac{n}{p}$. In this setting, balanced workloads make a difference (i.e., $T_b < T_u$) since P_f sends a smaller number of elements to P_s than in the unbalanced case.

6.2 One-to-all broadcast

Figure 4 (a) compares the execution time of the algorithm assuming the root node is either P_s or P_f. The plot demonstrates that their is neglible improvement in performance. The HBSP^k model predicted this behavior. The broadcast operation takes small advantage of the heterogeneity since each processor must receive all of the data. In fact, the improvement in performance is a result of P_f distributing $\frac{n}{p}$ integers to each processor during the first phase of the algorithm. Our analysis also applies if processor j receives $c_j n$ elements during phase one of the algorithm. Figure 4 (b) corroborates the theoretical results. Figure 5 plots the predictions of the cost model.

7. CONCLUSIONS

The HBSP^1 model offers a framework that promotes the development of distributed applications for heterogeneous clusters of workstations. HBSP^1 incorporates a small set of parameters that characterise the underlying heterogeneous platform. Efficient algorithmic execution results from nodes receiving a workload proportional to their computational and communication abilities, if applicable. For example, a close examination of the one-to-all broadcast operation demonstrates

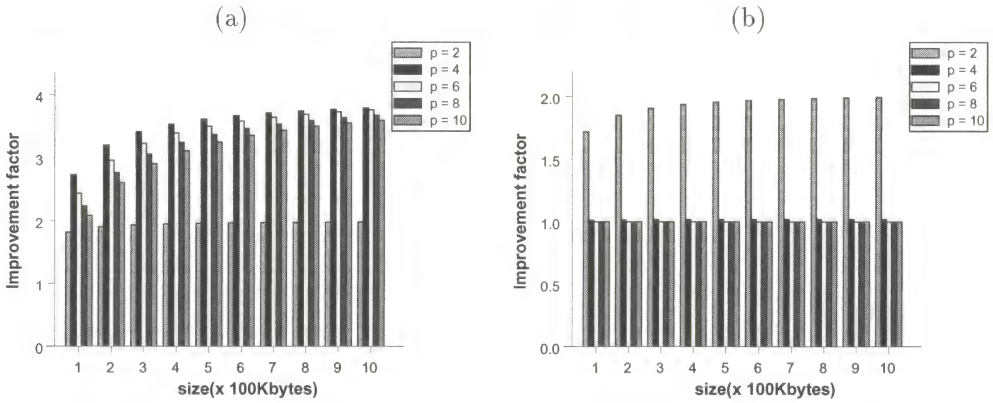


Figure 3: Scatter predicted performance. The improvement factor is determined by (a) $\frac{T_s}{T_f}$ and (b) $\frac{T_u}{T_b}$. The problem size ranges from 100KB to 1000KB of integers. Each data point represents the predicted performance on a cluster comprised of 2, 4, 6, 8 and 10 heterogeneous processors.

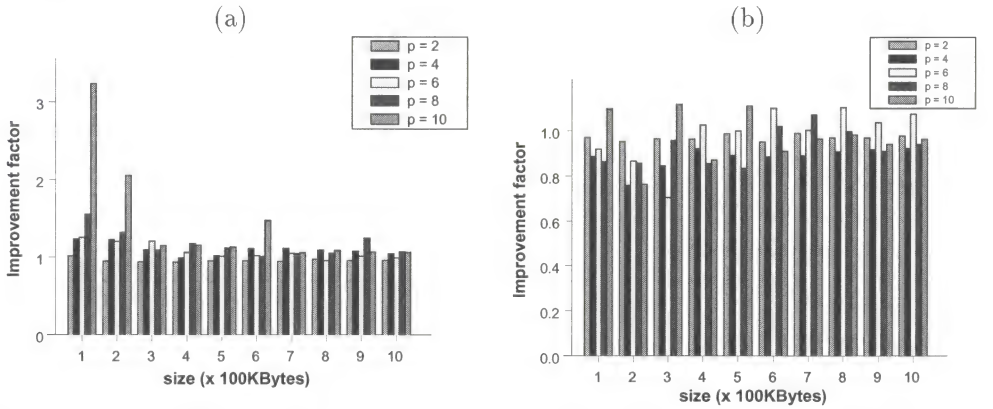


Figure 4: One-to-all braodcast actual performance. The improvement factor is determined by (a) $\frac{T_s}{T_f}$ and (b) $\frac{T_u}{T_b}$. The problem size ranges from 100KB to 1000KB of integers. Each data point represents the predicted performance on a cluster comprised of 2, 4, 6, 8 and 10 heterogeneous processors.

that it is impossible to avoid unbalanced workloads since the slowest machine must receive n items. The performance of our collective operations is quite impressive. Complete results are shown in Williams (2000). Fundamental changes to the algorithms are not necessary in order to attain an increase in performance. Besides good performance, the model predicts the behaviour of our collective routines within a reasonable margin of error. The accuracy of the cost function depends on the choices made in the implementation of HBSP*lib*. Thus, one source for inaccurate predictions may result from the shortcomings of the library implementation.

In conclusion, HBSP^1 offers a single-system image of a heterogeneous platform to the application developer. Under HBSP^1, improved performance results from taking advantage of the underlying heterogeneity. By hiding the non-uniformity of the underlying system from the application developer, the HBSP^1 model offers an environment that encourages the design of heterogeneous distributed software in an architecture-independent manner. Extensions to this work include designing HBSP^1 applications that can take advantage of our heterogeneous collective routines. We also intend to perform additional experiments on a heterogeneous cluster with a larger set of workstations.

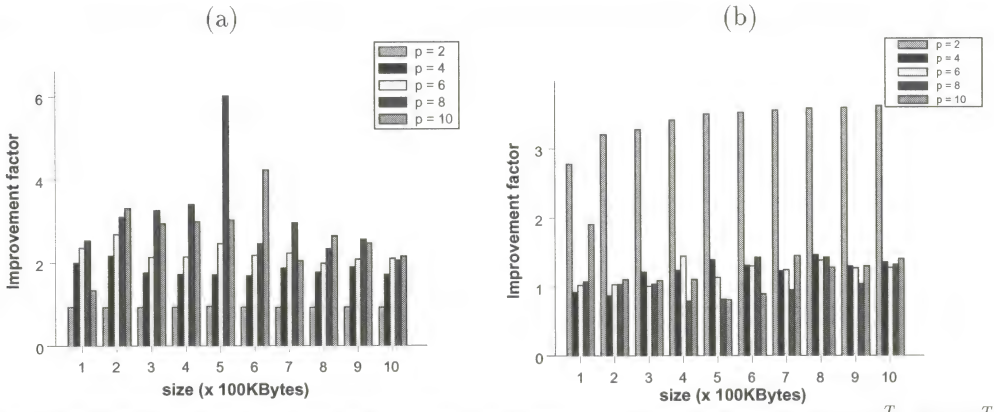


Figure 5: One-to-all braodcast predicted performance. The improvement factor is determined by (a)$\frac{T_s}{T_t}$ and (b)$\frac{T_u}{T_b}$. The problem size ranges from 100KB to 1000KB of integers. Each data point represents the predicted performance on a cluster comprised of 2, 4, 6, 8 and 10 heterogeneous processors.

REFERENCES

BANIKAZEMI, M., MOORTHY, V. and PANDA, D. (1998): Efficient collective communication on heterogeneous networks workstations, *International Conference on Parallel Processing,* 460-467.

BANIKAZEMI, M., SAMPATHKUMAR, J., PRABHU, S., PANDA, D. and SADAYAPPAN, P. (1999): Communication modeling of heterogeneous networks of workstations for performance characterisation of collective operations, *Heterogeneous Computing Workshop (HCW '99),* 125-133.

BHAT, P., RAGHAVENDRA, C. and PRASANNA, V. (1999): Efficient collective communication in distributed heterogeneous systems, *International Conference on Distributed Computing Systems.*

GERBESSIOTIS, A. V. and VALIANT, L. G. (1994): Direct bulk-synchronous parallel algorithms, *Journal of Parallel and Distribute Computing* 22(2): 251-267.

GOUDREAU, M. W., LANG, K., RAO, S. B., SUEL, T. and TSANTILAS, T. (1999): Portable and efficient parallel computing using the BSP model, *IEEE Transactions on Computers* 48(7): 670-689.

HILL, J. M. D., DONALDSON, S. R. and SKILLICORN, D. (1997): Portability of performance with the BSPlib communications library, *Programming Models for Massively Parallel Computers (MPPM '97).*

HILL, J. M. D., MCCOLL, B., STEFANESCU, D. C., GOUDREAU, M. W., LANG, K., RAO, S. B., SUEL, T., TSANTILAS, T. and BISSELING, R. (1998): BSPlib: The BSP programming library, *Parallel Computing* 24(14): 1947-1980.

JUURLINK, B. H. H. and WIJSHOFF, H. A. G. (1996): A quantitative comparison of parallel computation models, *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures,* 13-24.

LOWEKAMP, B. B. and BEGUELIN, A. (1996): Eco: Efficient collective operations for communicationon heterogeneous networks, *International Parallel Processing Symposium,* Honolulu, HI, 399-405.

MORIN, P. (1998): Coarse-grained parallel computing on heterogeneous systems, *Proceedings of the 1998 ACM Symposium on Applied Computing,* 629-634.

SUNDERAM, V. S. (1990): PVM: a framework for parallel distributed computing, *Concurency: Practice and Experience 2(4): 315-349.*

VALIANT, L. G. (1990): A bridging model for parallel computation, *Communications of the ACM* 33(8): 103-111.

WILLIAMS, T. L. (2000): *A General-Purpose Model for Heterogeneous Computation,* Ph.D. dissertation, University of Central Florida, Orlando.

WILLIAMS, T. L. and PARSONS, R. J. (2000): The heterogeneous bulk synchronous parallel model, *Parallel and Distributed Processing, Vol. 1800 of Lecture Notes in Computer Science*, Springer-Verlag, Cancun, Mexico, 102-108.

BIOGRAPHICAL NOTES

Tiffani Williams is a Visiting Lecturer of Computer Science at the University of Central Florida. She received a Ph.D. in Computer Science from the University of Central Florida. Her research interests are in heterogeneous distributed computing, models of computation, and computational biology. She is a member of the ACM and IEEE.

Rebecca Parsons is an Assistant Professor at the University of Central Florida. Her research interests are parallel computing, programming languages, and computational biology.

RBA: A Best Effort Resource Allocation Algorithm for Asynchronous Real-Time Distributed Systems*

B. Ravindran and T. Hegazy

The Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University, Blacksburg, VA 24061
E-mail: binoy@vt.edu, thegazy@vt.edu

We present a best effort resource allocation algorithm called RBA for asynchronous real-time distributed systems. The algorithm uses Jensen's benefit functions for expressing application timeliness requirements and proposes adaptation functions to describe the anticipated application workload during future time intervals. Furthermore, RBA considers an adaptation model where subtasks of application tasks may be replicated at run-time for sharing workload increases, and a real-time Ethernet system model where message collisions are deterministically resolved. Given such application, adaptation, and system models, the algorithm's objective is to maximise aggregate application benefit and minimise aggregate missed deadline ratio. Since determining the optimal allocation is computationally intractable, RBA heuristically computes the number of replicas that are needed for task subtasks and their processor assignment such that the resulting allocation is as "close" as possible to the optimal allocation. We also experimentally study RBA's performance under different scheduling and routing algorithms. The experimental results reveal that RBA produces higher aggregate benefit and lower missed deadline ratio under DASA than when the RED algorithm is used for scheduling and routing.

Categories and Subject Descriptors: Computer Systems Organisation – Computer-Communication Networks – Distributed Systems (C.2.4); Computer Systems Organisation – Special-Purpose and Application-Based Systems (C.3): Real-time and embedded systems; Software – Operating Systems – Process Management (D.4.1): Scheduling; Software – Operating Systems – Organisation and Design (D.4.7): Real-time systems and embedded systems; Software – Operating Systems – Organisation and Design (D.4.7): Distributed systems

General Terms: Algorithms, Experimentation, Measurement, Performance

Keywords: adaptive resource allocation, asynchronous distributed systems, benefit accrual model, best effort scheduling, real-time Ethernet, real-time systems, quality of service

1. INTRODUCTION

Real-time computer systems that are emerging for the purpose of strategic mission management such as collaborating within a team of autonomous entities conducting manufacturing, maintenance, or combat are "asynchronous" in the sense that processing and communication latencies do not

*Supported by the U.S. Office of Naval Research under Grant Number N00014-00-1-0549

Copyright© 2001, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 14 April, 2001, revised 11 August, 2001
Associate Editor: Michael Oudshoorn

necessarily have known upper bounds, and event and task arrivals may be non-deterministically distributed. This emerging generation of real-time distributed computing systems are very important for (real-time) supervisory control in many domains, including defense, industrial automation, and telecommunications (Jensen, 1992).

The vast majority of the past efforts on real-time computing focus on hard real-time systems that is usually centralised, but occasionally distributed (Buttazzo, 1997; Liu, 2000). The hard real-time computing theory assumes that application properties such as execution times and communication delays and execution environment characteristics are deterministically known with absolute certainty. The theory exploits such a-priori information and provides the hard real-time guarantee of "always meet all deadlines." However, the non-determinism inherent in the application and execution environment of asynchronous real-time distributed systems generally makes it non-cost-effective or even impossible to complete the execution of every real-time computation at its optimum time (e.g., before the deadline).

Recent advances in distributed systems research have produced quality of service (QoS) technologies such as Abdelzaher and Shin (1998); Brandt, Nutt, Berk and Mankovich (1998); Humphrey, Brandt, Nutt and Berk (1997); Hull, Shankar, Nahrstedt and Liu (1997); Ravindran (to appear); Rajkumar, Lee, Lehoczky and Siewiorek (1997); Rosu, Schwan, Yalamanchili and Jha (1997) that allow applications to specify and negotiate service expectations such as timeliness and accuracy, which were not previously possible under classical real-time theory. The QoS techniques consider application models where tasks can operate at multiple, discrete "levels" of service. A level is a strategy for doing a tasks' work and is characterised by a resource usage such as CPU utilisation, a QoS dimension such as accuracy of computational results, and a user-specified benefit. Therefore, given a set of QoS levels at which a task can operate, an adaptation mechanism can determine the "right level" of QoS depending upon the availability of internal resources to optimise a system-wide criteria such as aggregate task benefit.

In this paper, we advance the QoS technology by presenting a "best effort" resource allocation algorithm for asynchronous real-time distributed systems. The algorithm called Response Time Analysis-Based Best Effort Resource Allocation (RBA) is best effort in the sense that it seeks to maximise aggregate application benefit and minimise aggregate application missed deadline ratio, where the best possible benefit that the application can accrue is specified by the application itself.

We consider an application model that consists of trans-node periodic and aperiodic tasks that have end-to-end timeliness requirements, expressed using Jensen's benefit functions (Jensen, 1992). Further, we propose the concept of adaptation functions for describing the anticipated application workload during future time intervals. The functions are user-specified, can have arbitrary shapes, and may be dynamically modified to express the future application workload that is anticipated at a given time. Furthermore, we consider an adaptation model for the application, where subtasks of application tasks may be replicated at run-time for sharing workload increases, and a real-time Ethernet system model where message collisions are resolved using a deterministic routing algorithm.

Given such application, adaptation, and system models, our objective is to maximise the aggregate application benefit and to minimise the application missed deadline ratio during a future window of time with a known, but arbitrary workload pattern. This problem can be shown to be NP-complete (Hegazy, 2001). Thus, RBA is a polynomial-time heuristic algorithm that seeks to maximise the aggregate benefit and minimise the aggregate missed deadline ratio. To study how different scheduling and routing algorithms affect RBA's performance, we conduct benchmark-

driven simulation experiments. The experimental results reveal that RBA produces higher aggregate benefit and lower missed deadline ratio under DASA than when the RED algorithm is used for scheduling and routing.

Thus, the major contribution of the paper is the RBA algorithm that seeks to maximise aggregate benefit and minimise aggregate missed deadline ratio in asynchronous real-time distributed systems. To the best of our knowledge, we are not aware of any efforts that solve the problem solved by RBA.

The rest of the paper is organised as follows: Section 2 presents our application, adaptation, and system models. We discuss the objectives of the work and informally state the problem in Section 3. Section 4 discusses the heuristics employed by RBA, the rationale behind the heuristics, and each step of the algorithm in detail. Experimental evaluation of RBA is discussed in Section 5. Finally, the paper concludes with a summary of the work and its contributions in Section 6.

2. THE APPLICATION, ADAPTATION, AND SYSTEM MODELS

2.1 Application Model

We denote the set of tasks in the application by the set $T = \{T_1, T_2, T_3,...\}$, where a task can be either periodic or aperiodic. Each aperiodic task T_j has a "triggering" periodic task T_k that triggers its execution. After a periodic task T_k completes its execution, it may generate zero or more number of events that trigger the execution of the corresponding aperiodic task T_j.

The period of a task T_i is denoted as *period*(T_i), if T_i is periodic. If T_i is aperiodic, *period*(T_i) denotes the period of the periodic task that triggers task T_i. The periods of periodic tasks need not be the same and can be different for different tasks.

Each task T_i is assumed to consist of a set of subtasks (executable programs), which execute in a *serial* fashion. We use the notation $T_i = [st_1^i, m_1^i, st_2^i, m_2^i, ..., st_n^i, m_n^i]$ to represent a task T_i that consists of n subtasks and n messages to be executed in series. That is, subtask $st_i^k (i > 1)$ cannot execute before message m_{i-1}^k arrives. The set of subtasks of a task T_i is denoted as $ST(T_i) = \{st_1^i, st_2^i, st_3^i, ..., st_n^i\}$ and the set of inter-subtask messages of a task T_i is denoted as $MS(T_i) = \{m_1^i, m_2^i, m_3^i, m_4^i, ..., m_n^i\}$.

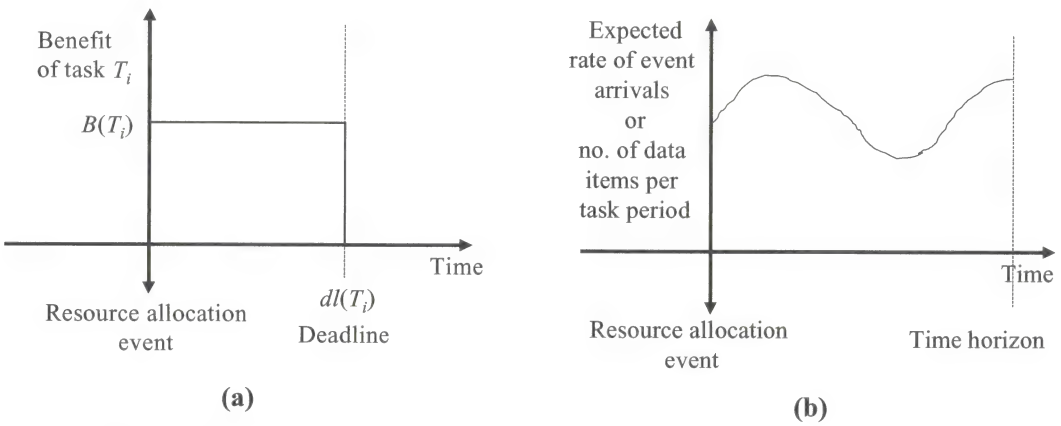


Figure 1. (a) A Benefit Function and (b) an Adaptation Function

We use Jensen's benefit functions for expressing application timeliness requirements (Jensen, 1992) and assume rectangular benefit functions for all tasks such as the one shown in Figure 1(a). Thus, completing a task anytime before its deadline will result in uniform benefit; completing it after the deadline will result in zero benefit. We denote the benefit of a task T_i —the height of T_i 's benefit function—as $B(T_i)$ and the deadline of a task T_i as, $dl(T_i)$, respectively.

We model the workload of a subtask and that of an inter-subtask message as the number of data items that it needs to process and transmit, respectively. Our motivation for this is due to the fact that the number of sensor reports and aperiodic events (or "data items") processed and transmitted by subtasks and messages, respectively, constitute the most significant part of the application workload in many real-time distributed applications that we regard as asynchronous (Welch, Ravindran, Shirazi and Bruggeman, 1998; Nswc, 1999). Furthermore, the major element of uncertainty in the processing and communication latencies in these systems is due to the uncertainty in the number of sensor objects and aperiodic events that the application has to process, as they are dependent upon the applications' external environment.

We assume that application-profile functions that can estimate subtask execution times as a function of data size workloads are available. Such execution time estimates can be regarded as a worst-case lower bound for processing a given data size workload. The profile functions can be determined by application profiling and measurement as described in Devarasetty (2001) and Hegazy (2001). We denote the lower bound on the estimated execution time of a subtask st_j^i for processing a data size d as $eex(st_j^i, d)$ and the estimated communication delay of an inter-subtask message m_j^i to transport a data size d as $ecd(m_j^i, d)$.

2.2 Adaptation Model

We assume an adaptation model for the application where subtasks of tasks can be *replicated* at run-time. The idea behind replication of subtasks is that once a subtask is replicated, replicas of the subtask can share the workload that was processed by the original subtask. Furthermore, concurrency can be exploited by executing the replicas on different processors and thereby the end-to-end task latency can be reduced. Thus, replication is allowed as a means to reduce task latencies when task workloads increase at run-time. For simplicity in the design of the application and the resulting application model, we assume that the workload of a subtask is equally distributed among all its replicas.

The state consistency of the subtask replicas is not considered in this work, as we assume that the tasks process data objects that are "continuous" in the sense that their values are obtained directly from a sensor in the application environment, or computed from values of other such objects. The replicas are thus assumed to be *temporally consistent* without applying every change in value, due to the continuity of physical phenomena.

We propose adaptation functions for expressing anticipated workload scenarios of the application during future time intervals. The adaptation functions describe the anticipated application workload as a function of the time (or a reference point such as task period) at which it is anticipated to occur. The functions can have arbitrary shapes and are user-specified for each application task. We regard the origin of the functions' axes as a "resource allocation event" in the sense of the start time of the expected scenario for the workload. The function is specified for a fixed duration of time into the future and ends at a time instant called the "time horizon." The anticipated workload scenarios—and thus the adaptation functions—may be dynamically modified, as the user's perception regarding the future workload changes. We regard adaptations functions to

be derived from the requirements and operational scenarios of the application. An example adaptation function is shown in Figure 1(b).

In this work, we use task periods as reference points for describing anticipated workloads. Thus, we denote the anticipated workload of a task T_i during a period p as $Adapt(T_i, p)$. For a periodic task, the anticipated workload is defined as the number of data items that is anticipated during a task period. For an aperiodic task, the anticipated workload is defined as the number of events that the triggering periodic task (of the aperiodic task) is anticipated to produce at the end of its period. The anticipated workload is assumed to be a constant during the task period, but may be different for different periods.

Note that the anticipated workload is not a necessity for resource allocation. If the anticipated workload is not available, resource allocation can still be done by using either (1) the workload that was observed during past time intervals or (2) a projected workload from the past workloads. Thus, it is important to note that resource allocation is independent of adaptation functions. If the functions are available, they will facilitate user-specified adaptation of the system. We believe that this "human-in-the-loop" approach with adaptation functions' concept will enable strategic allocation of resources that will yield greater system utility.

2.3 System Model

We consider a LAN-like distributed system where processors are interconnected through a router in a star-topology fashion. The router is equipped with a multi-port Ethernet adapter such as Znyx (2001). Each processor is connected to the router using a full-duplex Ethernet link (IEEE 802.3) and to a port at the router that is dedicated for the processor. Thus, the link between each processor and

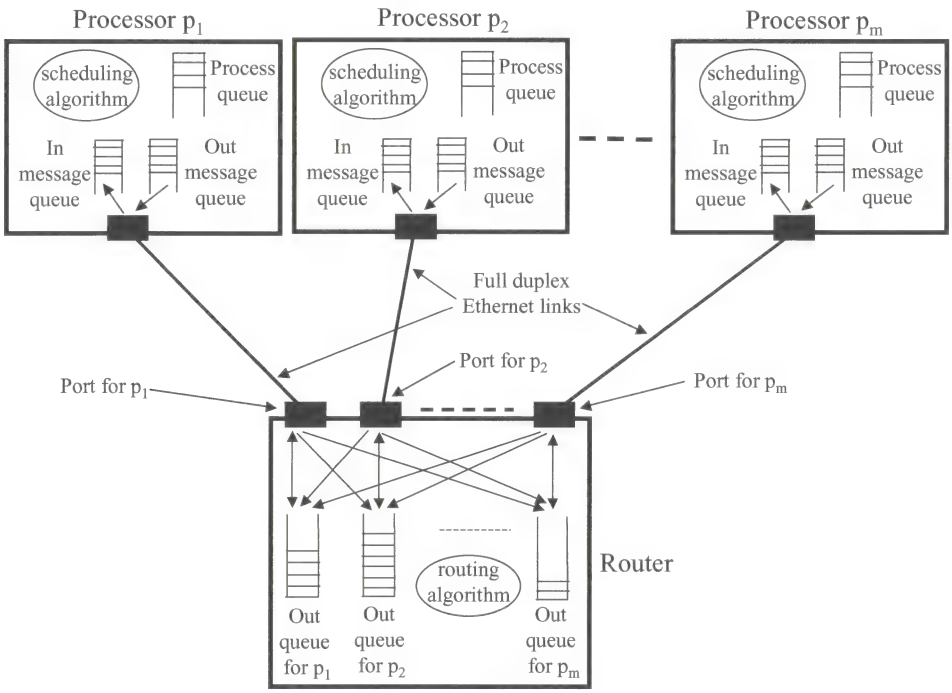


Figure 2. The Real-Time Ethernet Network System Model

the router becomes a dedicated link for simultaneous two-way communication between the processor and the router.

A message that is sent from an application subtask is first transmitted from the processor of the subtask to the router and then from the router to the processor of the destination subtask (Figure 2). The router, which maybe a processor, simply maintains a list of ready queues per processor and stores incoming messages that are destined for a processor on the processor's ready queue. When the outgoing link from the router to a processor becomes free for transmission, the router uses a routing algorithm that selects a message for transmission from the ready queue of the processor. Thus, messages collisions that occur due to access contention for the network is resolved using the deterministic routing algorithm.

The set of processors is denoted by the set $PR = \{p_1, p_2, p_3,..., p_m\}$. We assume that the clocks of the processors are synchronised using an algorithm such as Mills (1995). Furthermore, we use the non-preemptive version of the scheduling algorithm used at the processors for routing messages at the router. This is done for system homogeneity and the consequent simplicity that we obtain in the system model.

For scheduling and routing, we consider the classical EDF scheduling algorithm (Stankovic, Spuri, Ramamritham and Buttazzo, 1998) and best effort real-time scheduling algorithms such as DASA (Clark, 1990); LBESA (Locke, 1986); RED (Buttazzo and Stankovic, 1995). DASA, LBESA, and RED are known to outperform EDF during overloaded situations and perform the same as EDF during under-loaded situations where EDF is optimal.

3. THE OBJECTIVES AND PROBLEM STATEMENT

Given the application, adaptation, and system models described in Section 2, our objective is to maximise the aggregate task benefit and minimise the aggregate task missed deadline ratio during the time window of the adaptation functions. We define the aggregate task benefit as the sum of the benefit accrued by the execution of each task and the aggregate missed deadline ratio as the sum of the ratio of the number of tasks that missed their deadlines to the total number of tasks.

Thus, the problem that we are addressing in this paper can be informally stated as follows: "Given adaptation functions for each task in the application that may have arbitrary shapes and thus define an arbitrary task workload, what is the number of replicas needed for each task subtask and processors for executing them such that the resulting resource allocation will maximise the aggregate task benefit and minimise the aggregate task missed deadline ratio, during the time window of the adaptation functions?"

This problem can be shown to be an NP-complete problem (Hegazy, 2001). Thus, RBA is a heuristic algorithm that solves the problem in polynomial time, but not necessarily determine the number of subtask replicas and their processor assignment that will yield the maximum aggregate task benefit and minimum aggregate task missed deadline ratio. (The analysis of RBA's computational complexity can be found in Hegazy, 2001).

Besides determining a resource allocation that is as "close" as possible to the optimal allocation, we are also interested in determining the impact that different subtask scheduling and message routing algorithms have on the performance of the resource allocation algorithm. We experimentally evaluate this through a benchmark-driven simulation study.

4. THE RBA ALGORITHM: HEURISTICS AND RATIONALE

Since the objective of resource allocation is to maximise aggregate benefit and minimise aggregate missed deadline ratio, the desired properties of the RBA algorithm include:

1. *Allocate resources in the decreasing order of task benefits.* By doing so, we increase the possibility of maximising aggregate benefit, as the task selected next for resource allocation is always the one with the largest benefit among the unallocated tasks;
2. *Allocate resources for each task until its deadline is satisfied.* By seeking to satisfy the task deadline, we maximise the possibility of minimising the aggregate task missed deadlines. Furthermore, there is no reason to allocate resources for a task once it's deadline is satisfied, since the task benefit functions are rectangular and drop to zero benefit after the deadline;
3. *De-allocate resources for a task if it's deadline cannot be satisfied.* By doing so, we save system resources, which can be potentially used for satisfying deadlines of lower benefit tasks. This will increase the possibility of satisfying the deadlines of greater number of lower benefit tasks, resulting in potential contributions of non-zero benefit from them toward aggregate task benefit.

It is also important to note that when resources are being allocated for a task, we may reach a point before the satisfaction of the task deadline, after which any more increase in resources for the task will negatively affect the timeliness of higher benefit tasks, decreasing the aggregate task benefit that is accrued so far. At such points, it is not obvious what choice—whether to continue the allocation for the task or to stop and de-allocate—can yield higher aggregate benefit. For example, it may be possible that continuing the resource allocation for the task may eventually satisfy its deadline (at the expense of one or more higher benefit tasks). However, this may also satisfy the deadlines of greater number of lower benefit tasks resulting in greater aggregate task benefit than that would have been possible if we were to de-allocate the task and proceed to the next lower benefit task.

At such points of "diminishing returns," RBA makes the choice of de-allocating all resources allocated to the task. The rationale behind this choice is that since it is not clear how many higher benefit tasks will have to "pay" for satisfying the task deadline, it maybe best not to "disturb" the aggregate benefit that is accrued so far. Moreover, since resources are always allocated in decreasing order of task benefits, the chances of obtaining a higher aggregate benefit may be higher by satisfying as many high benefit tasks as possible.

RBA_BestEffortAlgorithm(T,W) /* T is the task set and W is the future time window */
1. Sort the task set in descending order according to task benefits;
2. For each task $i = 1$ to $|T|$ /* in decreasing order of task benefits */
 2.1 For each period $j = 1$ to $\lceil W / period(T_i) \rceil$
 2.1.1 For each subtask $k = 1$ to $|ST(T_i)|$
 DetermineReplicasAndProcessors$(st_k^i, j, Adapt(T_i, j))$;

Figure 3. Pseudo-code of the RBA Algorithm

Thus, RBA performs resource allocation according to the heuristic choices discussed here. We now summarise the algorithm as follows: RBA performs resource allocations at run-time when user-modifications to adaptation functions of application tasks are detected. Since the anticipated workload may be different for different task periods in the time window specified by the adaptation functions, RBA allocates resources for each period in the time window of the adaptation function,

starting from the most recent period and proceeding to the least recent. When triggered at run-time, the algorithm first sorts all tasks according to their benefits. For each task and for each adaptation period (in decreasing order of task benefits and period occurrences, respectively), RBA determines the number of replicas needed for each task subtask and their processor assignment that will satisfy the task deadline for the current period. While computing the number of subtask replicas for the task, if the timeliness of a higher benefit task is affected or if the task is found to be infeasible, the algorithm de-allocates all allocated replicas and proceeds to the next adaptation period. A high-level pseudo-code of the algorithm is shown in Figure 3.

Determining the number of replicas needed for each subtask of a task and their processor assignment that will satisfy the task deadline for a given adaptation period is non-trivial. The main difficulty lies in determining the end-to-end task response time given some number of replicas for each task subtask, as it requires holistic analysis of the system, which can be computationally intractable. To solve this problem, RBA employs the following important heuristic decision:

Assign deadlines to subtasks and messages of the task from the task deadline in such a way that if all subtasks and messages of the task can meet their respective deadlines, then the task will able to meet its deadline.

Based on this heuristic, we can determine the number of replicas needed for each task subtask that will satisfy the task deadline by solving the problem at the subtask-level i.e., by determining the number of replicas needed for each subtask that will satisfy the *subtask deadline*, as deadline satisfaction of all subtasks of the task is assumed to automatically satisfy the task deadline.

We now discuss the steps involved in determining the number of subtask replicas and their processor assignment. Section 4.1 discusses subtask and message deadline assignment. Estimation of subtask response time and message delays is discussed in Sections 4.2 and 4.3, respectively. Section 4.4 discusses how RBA computes the number of replicas and their processors.

4.1 Deadline Assignment of Subtasks and Messages

The problem of subtask and message deadline assignment from task deadlines has been studied in a different context (Kao and Garcia-Molina, 1997). The equal flexibility (EQF) strategy presented by Kao and Garcia-Molina (1997) assigns deadlines to subtasks and messages from the task deadline in a way that is proportional to their execution times and communication delays, respectively.

RBA uses EQF in the following way: The algorithm estimates subtask execution times and message communication delays using application-profile functions for a workload that is anticipated on the average. The estimated execution times and message delays are then used to assign subtask and message deadlines, according to EQF, respectively. Thus, the deadline of a subtask st_i^k for an assumed average data size of d_{avg} is given by:

$$dl\left(st_i^k\right) = eex\left(st_i^k, d_{avg}\right) + \left(dl\left(T_k\right) - \sum_{j=i}^{m} eex\left(st_j^k, d_{avg}\right) - \sum_{j=i+1}^{m} ecd\left(m_j^k, d_{avg}\right)\right) \times \left[eex\left(st_i^k, d_{avg}\right) \Big/ \left(\sum_{j=i}^{m} eex\left(st_j^k, d_{avg}\right) + \sum_{j=i+1}^{m} ecd\left(m_j^k, d_{avg}\right)\right)\right]$$

The deadline of a message m_i^k for an assumed average data size of d_{avg} is given by:

$$dl\left(m_i^k\right) = ecd\left(m_i^k, d_{avg}\right) + \left(dl\left(T_k\right) - \sum_{j=i}^{m} ecd\left(m_j^k, d_{avg}\right) - \sum_{j=i+1}^{m} eex\left(st_j^k, d_{avg}\right)\right) \times \left[ecd\left(m_i^k, d_{avg}\right) \Big/ \left(\sum_{j=i}^{m} ecd\left(m_j^k, d_{avg}\right) + \sum_{j=i+1}^{m} eex\left(st_j^k, d_{avg}\right)\right)\right]$$

4.2 Estimating Subtask Response Time

The response time of a subtask st_j^i of a task T_j under fixed priority schedulers is given by the classical equation $R_i^j = C_i^j + I_i^j$, where R_i^j is the subtask response time, C_i^j is the subtask execution time, and I_i^j is the interference that the subtask experiences from other subtasks (Audsley, Burns, Richardson, Tindell and Wellings, 1993). However, this equation is insufficient for best effort schedulers such as DASA and LBESA that we are considering in this work, as they make decisions at each scheduling event that are functions of the remaining subtask execution times at the event.

To determine the response time of a subtask on a processor, we first estimate the arrival times of all subtasks on the processor. To determine subtask arrival times, we assume the following:

1. Each periodic task arrives at the beginning of its period;
2. Each aperiodic task arrives at the end of the period of its triggering periodic task.

Thus, task jitters are assumed to be negligibly small. Though these assumptions are not realistic, we believe that they are reasonable for estimating the number of subtask replicas. Now, the arrival time of a subtask can be determined as the sum of the deadlines of all subtasks and messages that precede the subtask (under consideration) and the arrival time of the parent task of the subtask. Thus, given the arrival time of a task T_i, the arrival time of a subtask st_j^i of the task is given by:

$$arrival\left(st_j^i\right) = arrival\left(T_i\right) + \sum_{\forall k:k<j} \left[dl\left(st_k^i\right) + dl\left(m_k^i\right)\right]$$

Once the arrival times of subtasks are determined, RBA estimates the anticipated workload during each task adaptation period using the task adaptation functions. (For aperiodic tasks, the algorithm uses the period of their triggering periodic tasks as the task period.) The anticipated workloads are then "plugged into" the application profile functions to estimate the subtask execution times during the task periods.

The algorithm now estimates the subtask response times by determining the scheduling events that occur during the time window and by applying the scheduling algorithm at each scheduling event to determine the scheduling decision. Note that it is impossible to determine the subtask response times without determining the scheduling events (and decisions made at each event) for algorithms such as DASA and LBESA as their decisions at each event depends on the remaining subtask execution times at the event.

The pseudo-code of the subtask response time analysis algorithm is shown in Figure 4. The procedure *AnalyseResponse*, accepts a subtask *s*, a task period *p*, a processor *q* on which the response time of *s* needs to be determined, and the workload of the subtask as its arguments. Once the procedure determines the response times of all subtasks that are assigned to processor *q*, it then compares the subtask response times with the subtask deadlines. If all subtasks satisfy their deadlines, the procedure returns the response time of the subtask *s*. If any subtask is found to miss its deadline, the algorithm returns a "failure" value, indicating that replicating subtask *s* on processor *q* will affect the timeliness of higher benefit tasks.

Note that whenever the procedure *AnalyseResponse* is invoked for a subtask *s* for a processor *q*, all existing subtasks on *q* must belong to higher benefit tasks than the task of *s*, since RBA allocates replicas to tasks in decreasing order of their benefits.

The procedure *AnalyzeResponse* is independent of the scheduling algorithms used on the processors and at the router. The procedure only invokes an abstract function called *LocalScheduler* (Figure 4, line 9.4) that is responsible for selecting the next subtask (or message) from the process (or message) ready queue depending upon the algorithm used.

AnalyzeResponse(s,p,q,l) /* s is the subtask, p is the period number, q is the processor, l is the workload */

1. $ArrivalEvents[\]=\varnothing$; $Process[\]=\varnothing$;

/* Lists $ArrivalEvents[\]$ and $Process[\]$ have 3 fields: time, subtask, remaining_time */

2. $x=ParentTaskNumber(s)$; $y=SubtaskNumber(y,x)$; /* y is subtask # of s within its parent task x */

3. if $type(s)=PERIODIC$ /* $type(s)$ returns PERIODIC if s is periodic; otherwise returns APERIODIC */

 3.1 $ArrivalTime(s) = (p-1)\times period(T_x)+\sum_{r<y}\left(dl(st_r^x)+dl(m_r^x)\right)$;

 3.2 $StopTime=p\times period(T_x)$;

4. else /* s is aperiodic */

 4.1 $ArrivalTime(s) = p\times period(T_x)+\sum_{i<y}\left(dl(st_i^x)+dl(m_i^x)\right)$;

 4.2 $StopTime=(p+1)\times period(T_x)$;

5. $ArrivalEvents=ArrivalEvents\cup\left[st_y^x, ArrivalTime(s), eex(s,l)\right]$;

6. For each task $i = 1$ to $x-1$ /* consider all tasks that have higher benefits */

 6.1 For each period $j=1$ to $\lceil StopTime/period(T_i)\rceil$

 6.1.1 For each subtask $k=1$ to $|ST(T_i)|$

 6.1.1.1 if subtask st_k^i has a replica executing on processor q during period j

 1. $ArrivalTime(st_k^i) = p\times period(T_i)+\sum_{r<k}\left(dl(st_r^i)+dl(m_r^i)\right)$;

 2. if type$(st_k^i)=APERIODIC$ then $ArrivalTime(st_k^i) = ArrivalTime(st_k^i)+period(T_i)$;

 3. $ArrivalEvents=ArrivalEvents\cup\left[st_k^i, ArrivalTime(s), eex(s,l)\right]$;

7. $TerminationTime=\infty$; $SchedulerTime=0$; $CurrentProcess=1$;

8. Determine the earliest event $ArrivalEvents[k]$ from $ArrivalEvents$;

9. If $ArrivalEvents[k].time < TerminationTime$

 9.1 $Process[CurrentProcess].remaining_time = Process[CurrentProcess].remaining_time - ArrivalEvents[k].time - t$;

 9.2 $t=ArrivalEvents[k].time$;

 9.3 Add $ArrivalEvents[k]$ to $Process[\]$ and remove $ArrivalEvents[k]$ from $ArrivalEvents[\]$;

 9.4 $CurrentProcess=LocalScheduler(Process[\])$;

 9.5 $TerminationTime=t+Process[CurrentProcess].remaining_time$;

 9.6 return to step 8;

10. else

 10.1 $t=TerminationTime$;

 10.2 if $Process[CurrentProcess].subtask=s$ then $ResponseTime=t-ArrivalTime(s)$;

 10.3 if $t>Process[CurrentProcess].time+dl(Process[CurrentProcess].subtask)$

 10.3.1 return ∞; /*one of the subtasks of the higher benefit tasks missed its deadline*/

 10.4 Remove $Process[CurrentProcess]$ from $Process[\]$;

 10.5 return to step 8;

11. return $ResponseTime$;

Figure 4. Algorithm for Analyzing Subtask Response Time

4.3 Estimating Message Communication Delays

Estimating message communication delays in asynchronous real-time distributed systems is an intractable problem due to the large number of parameters that must be considered, requiring an exhaustive search. Therefore, RBA does not compute message delays during the resource allocation process. For the purpose of assigning message deadlines, RBA estimates message delays by

neglecting contentions that will occur at the router. Thus, the communication delay incurred for an inter-subtask message m_j^i to carry a data of size d is given by $ecd(m_j^i, d) = 2 \times (d/l_s)$, where l_s denotes the link transmission speed. Notice that we multiply by 2 since it takes two "hops" for the message to reach the destination from the source processor.

4.4 Determining Number of Subtask Replicas and Their Processors

RBA determines the number of subtask replicas and their processor assignment as follows: Let the subtask st_q^i be considered for replication, which currently has a single replica, denoted as, say $st_{q,0}^i$. The algorithm first analyses the response time of $st_{q,0}^i$ on its processor, say p_i. If the response time of $st_{q,0}^i$ is found to be less than the subtask deadline without affecting the timeliness of subtasks of higher benefit tasks, the algorithm concludes that the single replica $st_{q,0}^i$ is enough to satisfy its deadline and then proceeds to the next subtask.

If the response time of $st_{q,0}^i$ is larger than the subtask deadline or if executing $st_{q,0}^i$ on processor p_i makes one or more subtasks of higher benefit tasks miss their deadlines, the algorithm reduces the data size load of $st_{q,0}^i$ by replication. The algorithm considers a second replica for the subtask, denoted as, say $st_{q,1}^i$, which will reduce the data size load of the existing replica by half. To determine the processor for executing $st_{q,1}^i$, the algorithm analyses the subtask response time for processing half the data size on each of the processors, excluding p_i, as described in Section 4.2. The processor that gives the shortest response time, say p_j, is then selected. The algorithm now re-computes the response time of the first replica $st_{q,0}^i$ on processor p_i under half the data size, since $st_{q,1}^i$ will now process the other half of the data size. If response time of both the replicas is found to be shorter than the subtask deadline and execution of the replicas on the processors p_i and p_j is not found to affect the timeliness of higher benefit tasks, then two replicas are considered to be sufficient by the algorithm.

DetermineReplicasAndProcessors(s,i,l) /* s is the subtask, i is the period, and l is the anticipated load */

1. if $EstimateResponse(s,i,prr(s_0),l) < dl(s)$ return SUCCESS; /* done, if first replica is enough */
 /* s_0 is the first replica of s; $prr(s_0)$ is the processor that is executing s_0 */
2. $PT = PR - pr(s)$; /* $pr(s)$ is the set of processors that are executing replicas of s;
 PT is thus the set of processors that are not executing replicas of s */
3. if $PT = \varnothing$ return FAILURE;
4. For each processor $q \in PT$
 4.1 $ResponseTime = EstimateResponse(s,i,q,l/(|pr(s)+1|))$;
 /* $|pr(s)+1|$ is the number of replicas after increasing it by one and $l/(|pr(s)+1|)$ is the new load */
 4.2 Determine the minimum response time and the corresponding processor. Save the value of the minimum response time in *MinResponse* and the corresponding processor ID in $p_{\min}$;
5. $PT = PT - \{p_{\min}\}$;
6. $pr(s) = pr(s) \cup \{p_{\min}\}$;
7. if $MinResponse > dl(s)$ return to step 3;
8. For each processor $q \in pr(st_j^i) - \{p_{\min}\}$
 8.1 if $EstimateResponse(s,i,q,l/|pr(s)|) > dl(s)$ return to step 3;
9. return SUCCESS;

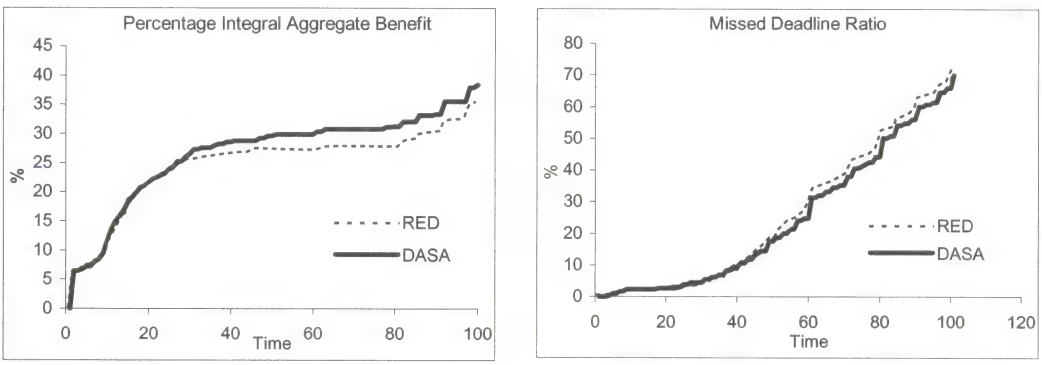
Figure 5. Algorithm for Determining Number of Subtask Replicas and their Processors

RBA repeats the process until each replica is able to satisfy the subtask deadline. Every time the algorithm considers adding a new replica, it checks whether the existing ones will be able to satisfy their deadlines without affecting the timeliness of higher benefit tasks. Note that as the number of replicas increases, the workload share of each replica will be reduced. If the algorithm determines that executing the maximum number of replicas for a subtask (which is equal to the number of processors in the system, for exploiting maximum concurrency) does not satisfy the subtask deadline, it de-allocates the task as discussed in Section 4.

Figure 5 shows the pseudo-code of the algorithm that determines the subtask replica sizes and their processor assignment. The procedure *DetermineReplicasAndProcessors* accepts a subtask s, a period i, an anticipated workload l during the period i, and determines the number of replicas for s and their processors. Recall that the procedure *RBA_BestEffortAlgorithm* shown in Figure 3 invokes the procedure *DetermineReplicasAndProcessors*.

5. EXPERIMENTAL EVALUATION

The baseline parameters of our simulation experiments were derived from the real-time benchmark described in Welch and Shirazi (1999). Details of the parameters can be found in Hegazy (2001).



(a) (b)

Figure 6. RBA's (a) Aggregate Accrued Benefit and (b) Aggregate Missed Deadline Ratio

Figure 6(a) shows the aggregate task benefit accrued by RBA under DASA and under RED when DASA and RED were used for scheduling and routing on all processors and at the router, respectively. Figure 6(b) shows the corresponding aggregate task missed deadline ratio. The metrics are shown as a function of time for the length of the task adaptation functions during which the anticipated workload of all tasks increases linearly. We observed that RBA's performance under DASA and under RED were similar to that under LBESA and under EDF, respectively. Hence, we omit the plots for LBESA and EDF. From the figures, we observe that RBA yields higher aggregate task benefit and lower aggregate task missed deadline ratio under DASA (and LBESA) than that under EDF and RED.

We also studied the performance of RBA during situations when the actual workload is different from the anticipated workload given by the adaptation functions. To characterise the algorithm's performance during such situations, we define a *relative load error* term as follows:

$$e_r = \frac{(anticipated\ load - actual\ load)}{anticipated\ load}$$

Figure 7 shows RBA's performance under a range of relative load errors from –0.9 to +0.9, for a fixed anticipated workload. A load error of 0.9 means that the actual load is 190% of the anticipated load. The y-axis shows the *relative change in aggregate benefit*. We define the *change in aggregate benefit* for a given value of e_r as the difference between the aggregate benefit under e_r and the aggregate benefit under zero relative load error. The *relative* change in aggregate benefit is defined as the ratio of change in aggregate benefit to aggregate benefit under zero relative load error.

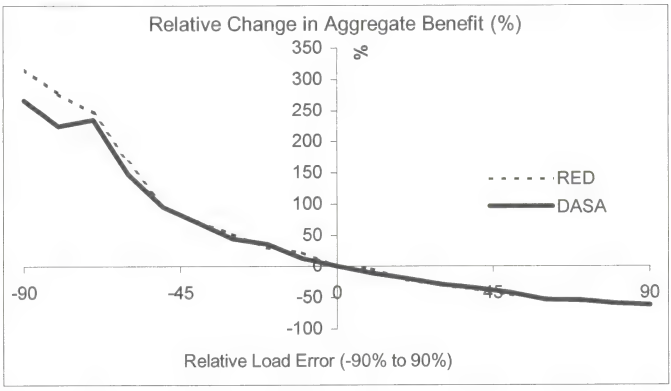


Figure 7. Effect of Error in Anticipated Load on RBA's Performance

From Figure 7, we observe that as the relative load error increases in the negative direction (i.e., when the actual load becomes smaller than the anticipated load), the aggregate benefit increases significantly and the rate of change is relatively high. On the other hand, as the relative load error increases in the positive direction (i.e., when the actual load becomes larger than the anticipated load), the aggregate benefit decreases with a relatively small rate of change. This illustrates the stability of the algorithm when the user-specified anticipated workload under-estimates the actual workload. We observed a similar pattern for all the scheduling algorithms that were considered. Thus, in Figure 7, we only show the performance of RBA under DASA and RED.

6. CONCLUSIONS

In this paper, we present a resource allocation algorithm called RBA for asynchronous real-time distributed systems. The algorithm considers an application model where application timeliness requirements are expressed using Jensen's benefit functions and proposes adaptation functions to describe the anticipated application workload during future time intervals. Furthermore, RBA considers an adaptation model where subtasks of application tasks are replicated at run-time for sharing workload increases, and a real-time Ethernet system model where message collisions are resolved using a deterministic routing algorithm. Given such application, adaptation, and system models, the objective of the algorithm is to maximise aggregate application benefit and minimise aggregate missed deadline ratio.

RBA solves this problem using a set of heuristics that compute the number of replicas needed for task subtasks and their processor assignment, in polynomial time, but not necessarily result in

the optimal allocation. To investigate the performance of RBA under different scheduling and message routing algorithms, we conduct benchmark-driven simulation experiments. The experimental results reveal that RBA produces higher aggregate benefit and lower missed deadline ratio under DASA than that under RED. We also observe that as the actual workload becomes larger than the anticipated workload, the aggregate benefit produced by RBA decreases with a relatively small rate of change.

Thus, the major contribution of the paper is the RBA algorithm that seeks to maximise aggregate application benefit and minimise aggregate missed deadline ratio in real-time distributed systems that are subject to workload uncertainties. Minor contributions include the concept of adaptation functions and the real-time Ethernet architecture.

REFERENCES

AUDSLEY, N., BURNS, A., RICHARDSON, M., TINDELL, K. and WELLINGS, A. (1993): Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal* 8(5): 284-292.

ABDELZAHER, T. and SHIN, K. G. (1998): End-host architecture for QoS-Adaptive communication. *Proceedings of The IEEE Real-Time Technology and Applications Symposium* (4): 121-130, IEEE CS Press.

BUTTAZZO, G. C. (1997): Hard real-time computing systems: Predictable scheduling algorithms and applications. Kluwer Academic Publishers.

BUTTAZZO, G. C. and STANKOVIC, J. A. (1995): Adding robustness in dynamic preemptive scheduling. In *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*. FUSSEL, D. S. and MALEK, M. (eds). Kluwer Academic Publishers.

BRANDT, S., NUTT, G., BERK, T. and MANKOVICH, J. (1998): A dynamic quality of service middleware agent for mediating application resource usage. *Proceedings of The IEEE Real-Time Systems Symposium* 19:307-317, IEEE CS Press.

CLARK, R. K. (1990): Scheduling dependent real-time activities. Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University.

DEVARASETTY, R. (2001): Heuristic algorithms for adaptive resource management of periodic tasks in soft real-time distributed systems. M.S. Thesis, The Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University.

HEGAZY, T. (2001): Using application benefit for proactive resource allocation in asynchronous real-time distributed systems. M.S. Thesis, The Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University.

HUMPHREY, M., BRANDT, S., NUTT, G. and BERK, T. (1997): The DQM architecture: middleware for application centered QoS resource management. *Proceedings of The IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*.

HULL, D., SHANKAR, A., NAHRSTEDT, K. and LIU, J. W. S. (1997): An end-to-end QoS model and management architecture. *Proceedings of The IEEE Workshop on Middleware for Distributed Real-Time Systems and Services* pages 82-89.

JENSEN, E. D. (1992): Asynchronous decentralised real-time computer systems. In *Real-Time Computing, Proceedings of the NATO Advanced Study Institute*. STOYENKO, A. D. and HALANG, W. (eds). St. Martin, Springer-Verlag.

KAO, B. and GARCIA-MOLINA H. (1997): Deadline assignment in a distributed soft real-time system. *IEEE Transactions on Parallel and Distributed Systems* 8(12): 1268 –1274.

LOCKE, C. D. (1986): Best-effort decision making for real-time scheduling. Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University.

LIU, J. W. S. (2000): Real-time systems. Prentice Hall.

MILLS, D. L. (1995): Improved algorithms for synchronising computer network clocks. *IEEE/ACM Transactions on Networks* 3: 245-254.

NSWC (1999): High performance distributed computing. The U.S. Naval Surface Warfare Center, Available at http://www.nswc.navy.mil/hiperd/index.shtml.

RAVINDRAN, B.and HEGAZY. T. (2001): A predictive algorithm for adaptive resource management of periodic tasks in asynchronous real-time distributed systems. *Proceedings of The IEEE International Parallel and Distributed Processing Symposium*, IEEE CS Press.

RAJKUMAR, R., LEE, C., LEHOCZKY, J. AND SIEWIOREK, D. (1997): A resource allocation model for QoS management. *Proceedings of The IEEE Real-Time Systems Symposium* (18): 298-307, IEEE CS Press.

ROSU, D., SCHWAN, K., YALAMANCHILI, S. and JHA, R. (1997): On adaptive resource allocation for complex real-time applications. *Proceedings of The IEEE Real-Time Systems Symposium* 18: 320-329, IEEE CS Press.

RAVINDRAN, B. (to appear): Engineering dynamic real-time distributed systems: Architecture, System Description Language, and Middleware. *IEEE Transactions on Software Engineering,* Available at: http://www.ee.vt.edu/~binoy/papers.html.

STANKOVIC, J. A., SPURI, M., RAMAMRITHAM, K. and BUTTAZZO, G. C. (1998): Deadline scheduling for real-time systems. Kluwer Academic Publishers.

WELCH, L. R. AND SHIRAZI, B. (1999): A dynamic real-time benchmark for assessment of QoS and resource management technology. *Proceedings of The IEEE Real-Time Technology and Applications Symposium* (5): 36-45, IEEE CS Press.

WELCH, L. R., RAVINDRAN, B., SHIRAZI, B. and BRUGGEMAN, C. (1998): Specification and modeling of dynamic, distributed real-time systems. *Proceedings of The IEEE Real-Time Systems Symposium* (19): 72-81, IEEE CS Press.

ZNYX (2001): The Znyx 340Q series multichannel PCI fast ethernet adapters. Znyx Networks Inc., Available at: http://www.znyx.com/products/netblaster/zx340q.htm.

BIOGRAPHICAL NOTES

Binoy Ravindran is an Assistant Professor in the Bradley Department of Electrical and Computer Engineering at Virginia Tech. He received his B.S. degree in Mechanical Engineering from the University of Kerala, India in 1991, M.S. degree in Computer Science from New Jersey Institute of Technology (NJIT) in 1994, and Ph.D. in Computer Science and Engineering from the University of Texas at Arlington (UT Arlington) in 1998. He received two "Student Achievement Awards" from NJIT for outstanding research by a graduate student (1994, 1995), and a "Doctoral Dissertation Award" from UT Arlington for outstanding doctoral research (1998). His current principal research focus is on asynchronous real-time distributed systems having custom, application-level, end-to-end quality of services (e.g., timeliness, survivability). During 1999, Dr. Ravindran was an invited speaker at INRIA, France and at the ARTES (Swedish National Strategic Research Initiative in Real-Time Systems) Real-Time Week, Sweden. He served as the Program Chair of the 1999 IEEE International Workshop on Parallel and Distributed Real-Time Systems. Currently, Dr. Ravindran is a Guest-Editor for IEEE Transactions on Computers for a 2002 Special Issue on "Asynchronous Real-Time Distributed Systems."

Tamir Hegazy is an M.S. student in Computer Engineering at the Bradley Department of Electrical and Computer Engineering at Virginia Tech. He received his B.S. degree in Electronic Engineering from Mansoura University, Egypt in 1997. He is the recipient of an "Honor Degree" and an "Excellence Prize" from Mansoura University for distinguished academic performance during the B.S. degree. His current research focus is on asynchronous real-time distributed systems. He has several publications on the topic of adaptive resource allocation in asynchronous real-time distributed systems in IEEE journals and conferences.

Multi-criterion Optimisation of Distributed System Performance by Evolutionary Task Assignments

Jerzy Balicki

Computer Science Laboratory
The Polish Naval Academy
ul. Smidowicza 69, 81-103 Gdynia, Poland
E-mail: jbalicki@amw.gdynia.pl

A careful task assignment in a distributed computer system may reduce the workload of a bottleneck computer. It can decrease the cost of computation if the computer sort selection is applied with the task assignment. The total system performance is another measure that can be minimised by the computer sort selection with the task assignment. An extended problem of task allocation can be formulated as a multiobjective combinatorial optimisation question, which is solved by an adaptive evolutionary algorithm. It is applied by finding the subset of Pareto-optimal solutions. Then, a module scheduling is used to maximise the probability of completing tasks with timing correctness.

CR Categories: Distributed systems, multiobjective optimisation, evolutionary algorithm

INTRODUCTION

An optimal task assignment in a distributed computer system may reduce the total cost of a program execution or the workload of a bottleneck computer (Bokhari, 1987). In the case of homogeneous systems, the total interprocessor communication time can be decreased by a rational allocation of tasks among computers (Lo, 1988). An approximation algorithm for minimisation of the turn-around time in heterogeneous distributed computing systems has been recently prepared (Kafil and Ahmad, 1998).

The timing correctness for each task is exceptionally significant in real-time systems. To incorporate timing correctness for the task system within the planning cycle, the precedence constraint has been introduced as well as the probability of meeting task deadlines (Hou and Shin, 1997). If the task is divided across some procedures (modules), a task flow graph can be prepared. It requires the use of a module scheduling algorithm to order the procedures assigned to each computer so that tasks may be completed on time (Weglarz, 1998).

Computers can be selected to decrease the time of module execution (Balicki, 1999). The workload of a bottleneck computer, the probability of meeting task deadlines as well as the cost of computers can be reduced. Moreover, the numerical performance of the system may be increased by the sensible computer sort assignment. Because the above criteria are in disagreement, a multiobjective optimisation problem is formulated (Choi and Wu, 1998). Another multi-criterion optimisation problem of task assignments has been discussed in (Balicki and Kitowski, 1999),

Copyright© 2001, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 13 February 2001, revised 8 August, 2001
Associate Editor: Michael Oudshoorn

where the total computer cost has been minimised with the overhead time of a distributed serial program execution.

In this paper, a multicriteria approach is intended for the design of an extended task allocation subject to precedence constraints and computer resource limits. The extended task allocation includes the task assignment to computers as well as the task schedule in each computer. Furthermore, an assignment of computers is considered. A new set of four evaluation criteria is discussed. A workload of the bottleneck computer, the probability of meeting task deadlines, the total computer cost, and a total amount of computer performance are the considered criteria for finding Pareto-optimal solutions.

In recent years, there has been significant interest in the use of modern metaheuristic multicriteria optimisation methods such as simulated annealing (Jaszkiewicz, 2000), tabu search (Hansen, 1997) or Hopfield models of neural networks (Balicki, 1999). Evolutionary calculations deal simultaneously with a population of possible solutions, which allows finding a subset of Pareto optimal solutions by one algorithm run instead of several separate runs of standard multiobjective optimisation techniques. An evolutionary approach is convenient for the above reason, if we look for the subset of Pareto-optimal solutions. We recommend an evolutionary algorithm with the two-weight binary tournament for finding the efficient extended task assignments.

A task system with precedence constraints has been applied for modelling of program modules written in the programming language C. These programs support some navigation tasks on a ship. Modules communicate by the local area network Ethernet that connect computers working under the QNX operating system. For finding the Pareto-optimal task assignment, a proposed adaptive evolutionary algorithm has been developed to design a distributed application.

2. COMPLETING TASKS WITH TIMING CORRECTNESS

We assume an invocation of periodic task repeats itself throughout the planning cycle. A periodic task is called at the fixed time intervals. The execution time, the required resources, and the invocation period L_n characterise the task T_n. The planning cycle of repeated tasks is the least common multiple L of periods for all tasks $\{ L_1,\ldots, L_n,\ldots, L_N \}$. It is the time interval $[t_0+kL; t_0+(k+1)L]$ for a nonnegative number of the planning cycle k. During each planning cycle, task T_n is started after an invocation time λ_n and it should be terminated no later than the deadline δ_n.

A task flow graph is commonly used to describe the logical structure of tasks consisted of modules as well as the communications and precedence constraints among modules (Hou and Shin, 1997). In an OR-subgraph that is a part of a task flow graph, the first output branch is taken with the given probability q and the second one with probability $(1–q)$ (Figure 1). In a loop, the given probability for repeating a module in a loop is denoted as l. The number of times when a module in a loop can be executed is no more than L_{max}.

A message passing allows modules to pass data to each other and provides means of synchronising the execution of cooperated modules. If module m_v of task T_n issues *send* the message to the module m_u of task T_r, T_n remains blocked until m_u issues *replay*. Then, the module m_u completes the processing associated with the *received* message from the module m_v. Afterwards, the replay message is sent to the module m_v. Both the tasks T_n and T_r are ready to run. They may be concurrently executed on different computers. If these modules are allocated to the same computer, which process runs depends on the relative priorities of modules.

A form of non-blocking message is called *a proxy* that does not require a reply of the receiving module. A destination process is not blocked in this case. If any event occurs or some date is

available for processing, a proxy may be sent to the module without acknowledgment from it. Signals can be considered as another method of asynchronous communication. No data is transferred with the signal. Signals are usually used to terminate or raise any process.

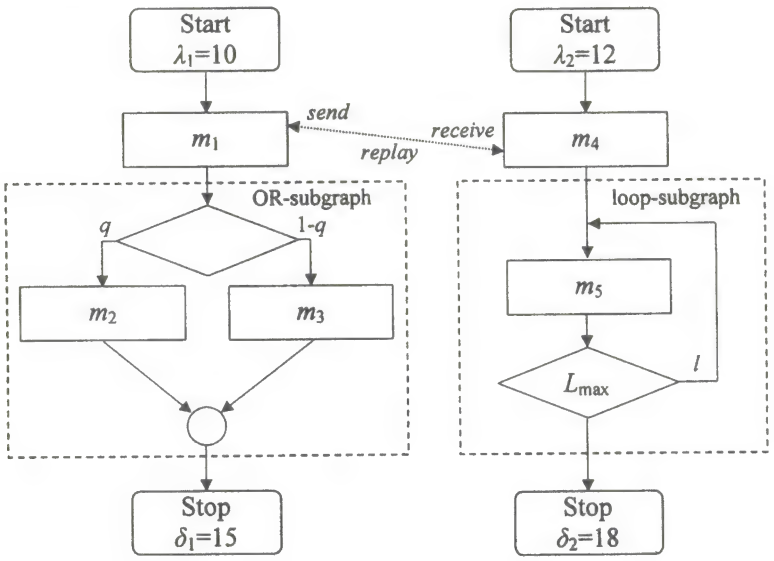


Figure 1. An example of task flow graph for 2 tasks and 5 modules

3. EVALUATION CRITERIA FOR EXTENDED TASK ASSIGNMENT

A numerical complexity of task assignment problems for the minimisation of the total cost of a distributed program execution depends on the number of computers, a computer memory limit, and the structure of an intermodule communication. There is an efficient network flow algorithm for a two-computer system (Stone, 1977). However, if the number of computers is greater than three or computer memory is limited, then the above problem is NP-hard (Sinclair, 1988). If the structure of an intermodule communication for a large-scale distributed computer system has a representation like a tree graph or the parallel-sequence graph and there is no memory limit, then the efficient algorithms based on the shortest path procedure can be exploited (Towsley, 1986).

Chu and Lan (1987) have introduced the workload of the bottleneck computer as the criterion for the evaluation of an allocation quality. A computer with the heaviest task load is the bottleneck machine in the system, and its workload is a critical value that should be minimised. The weight of the bottleneck computer is calculated, as below:

$$Z_{\max}(x) = \max_{i \in \overline{1,I}} \left\{ \sum_{j=1}^{J} \sum_{v=1}^{V} t_{vj} x_{vi}^{m} x_{ij}^{\pi} + \sum_{\substack{v=1 \\ u \neq v}}^{V} \sum_{\substack{u=1 \\ k \neq i}}^{V} \sum_{i=1}^{I} \sum_{k=1}^{I} \tau_{vuik} x_{vi}^{m} x_{uk}^{m} \right\} \quad (1)$$

where

t_{vj} – the overhead execution time of the module m_v at the computer πj,

τ_{vuik} – the total communication time between the module m_v allocated to the node w_i and the module m_u allocated to the node w_k,

$$x=[x_{11}^m,...,x_{1i}^m,...,x_{1I}^m,...,x_{vi}^m,...,x_{VI}^m,x_{11}^\pi,...,x_{1j}^\pi,...,x_{1J}^\pi,...,x_{ij}^\pi,...,x_{I1}^\pi,...,x_{Ij}^\pi,...,x_{IJ}^\pi]^T,$$

$$x_{ij}^\pi=\begin{cases}1 \text{ if } \pi_j \text{ is assigned to the } w_i,\\ 0 \text{ in the other case,}\end{cases} \quad i=\overline{1,I},\ j=\overline{1,J},$$

$$x_{vi}^m=\begin{cases}1 \text{ if } m_v \text{ is assigned to } w_i,\\ 0 \text{ in the other case,}\end{cases} \quad v=\overline{1,V},\ i=\overline{1,I}.$$

The other measure of the task assignment is a cost of computers that can be determined according to the subsequent formula:

$$F_2(x)=\sum_{i=1}^{I}\sum_{j=1}^{J}\kappa_j x_{ij}^\pi \tag{2}$$

where κ_j represents the cost of the computer π_j.

The third measure of the task assignment is a total amount of computer numerical performance that can be calculated according to the following formula:

$$\widetilde{F}_2(x)=\sum_{i=1}^{I}\sum_{j=1}^{J}\vartheta_j x_{ij}^\pi \tag{3}$$

where ϑ_j represents the numerical performance of the computer π_j.

Figure 2 shows a module assignment to computers for the given set of modules and a set of computer sorts. In the node w_i , a module m_v is assigned to the computer π_j and a module m_u is assigned to the computer π_1 in the node w_k. If there are some interactions between these modules, then the communication time τ_{vuik} is included to the workload of the computers in the nodes w_i and w_k. Moreover, it may change the load of the bottleneck computer. The computer π_j in the node w_i can decrease the total cost of computers F_2, if it has a higher numerical performance ϑ_j according to the assumed performance benchmark.

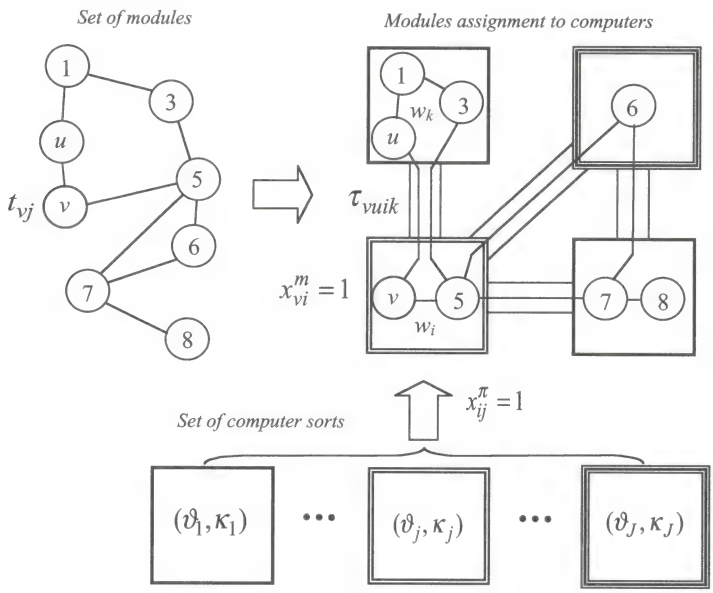


Figure 2. A module assignment to computers for a set of intercommunicated tasks and the set of different computers

4. MULTIOBJECTIVE OPTIMISATION PROBLEM

Let $(\mathcal{X}, F, P)$ be the multiobjective optimisation problem for finding the Pareto-optimal solution subset. It can be established, as follows:

1) $\mathcal{X}$ - an admissible solution set

$$\mathcal{X} = \{x \in \mathcal{B}^{I(V+J)} \mid \sum_{i=1}^{I} x_{vi}^{m} = 1, v = \overline{1,V}; \sum_{j=1}^{J} x_{ij}^{\pi} = 1, i = \overline{1,I};$$

$$\sum_{v=1}^{V} c_{vr} x_{vi}^{m} \le \sum_{j=1}^{J} d_{jr} x_{ij}^{\pi}, i = \overline{1,I}, r = \overline{1,R}\}, \ \mathcal{B} - \text{the set } \{0, 1\}$$

2) F - a vector quality criterion

$$F : \mathcal{X} \to \mathcal{R}^3 \tag{4}$$

where $F(x) = [Z_{\max}(x), F_2(x), -\tilde{F}_2(x)]^T$ for $x \in \mathcal{X}$

$Z_{\max}(x)$, $F_2(x)$ and $\tilde{F}_2(x)$ are calculated by (1), (2), (3), respectively,

3) P - the Pareto relationship

The relationship P for finding Pareto-optimal solutions is a subset of $Y \times Y$ where $Y=F(\mathcal{X})$. If $a \in Y$, $b \in Y$, and $a_n \le b_n, n = \overline{1,N}$, then $(a,b) \in P$. The definition of the Pareto relationship respects the minimisation of all criteria. There is no task allocation $a \in X$ such that $(F(a),F(x^*)) \in P$ for the Pareto-optimal assignment $x^* \in X, a \ne x^*$.

Each admissible solution is supposed to satisfy three classes of constraints. We assume one and only one computer should be allocated in each node. It implies the computer allocation constraints, as follows:

$$\sum_{j=1}^{J} x_{ij}^{\pi} = 1, i = \overline{1,I} \tag{5}$$

Because each unit is allocated to one node, then the task allocation constraints are formulated, as below:

$$\sum_{i=1}^{I} x_{vi}^{m} = 1, v = \overline{1,V} \tag{6}$$

Each computer should be equipped with necessary resource amounts for a program execution. Let the following memories $z_1,...,z_r,...,z_R$ be available in an entire system and let d_{jr} denote the capacity of memory z_r in the workstation π_j. The value d_{jr} is nonnegative and limited. We assume the module m_v reserves c_{vr} units of memory z_r and holds it during a program execution. The value c_{vr} is nonnegative and limited, too.

The memory limit in any machine cannot be exceeded in the ith node, as written below:

$$\sum_{v=1}^{V} c_{vr} x_{vi}^{m} \le \sum_{j=1}^{J} d_{jr} x_{ij}^{\pi}, \quad i = \overline{1,I}, \ r = \overline{1,R} \tag{7}$$

5. MULTI-CRITERION EVOLUTIONARY ALGORITHMS

An overview of evolutionary algorithms for multiobjective optimisation problems is submitted in Zitzler, Deb, and Thiele (2000). Some specific knowledge about the considered optimisation problem is applied in an evolutionary algorithm. Consequently, a standard genetic algorithm can be used for solving the wide class of optimisation problems, and each evolutionary algorithm is rather focused on the special class of problems. However, outcomes are regularly much better for evolutionary algorithms than for genetic algorithms (Michalewicz, 1996).

In the first multi-criterion genetic algorithm called a vector evaluated genetic algorithm VEGA, a population of solutions is divided on N subpopulations, where N is the number of partial criteria (Schaffer, 1985). For each nth subpopulation, the criterion F_n is a fitness function. However, a crossover and a mutation are carried out for the entire population. This method for a fitness evaluation has a weakness related with the discrimination of Pareto solutions situated in an interior of the Pareto front.

Fourman (1985) has considered the selection with binary tournaments, where two randomly chosen solutions have been compared. The hierarchical alternative is chosen and it is included to a mating pool of potential parents. A selection probability is calculated for the most significant aim. A random choice is carried out twice according to the roulette rule. Hierarchical tournaments push the population towards lexicographical solutions likewise the VEGA approach. Another selection is based on a random choice of a goal that is taken to comparison of selected solutions. Selection probabilities are constant or they can depend on the chosen purpose for the other tournament selection.

A ranking idea for non-dominated individuals has been introduced to avoid the prejudice of the interior Pareto alternatives by Goldberg (1989). Srinivas and Deb (1994) have built a non-dominated sorting genetic algorithm NSGA on the ideas mentioned by Goldberg (1989). If some admissible solutions are in a population, then the Pareto-optimal individuals are determined, and after that they get the rank 1. Subsequently, they are temporary eliminated from the population. Next, the new Pareto-optimal alternatives are found from the reduced population and they get the rank 2. In this procedure, the level is increased and it is repeated until the set of admissible solutions is exhausted. All non-dominated individuals have the same reproduction fitness because of the equivalent rank.

To maintain the diversity of the population and to preclude premature convergence, fitness-sharing techniques have been developed (Fonseca and Fleming, 1995). A mating restriction assumes that individuals from a criteria space neighbourhood are similar, so that they can form stable niches. If a non-dominated evaluation for the current population has a long distance to the nearest non-dominated evaluation and there is a niche of non-dominated results, then the separated individual should be preferred by increasing its fitness before individuals from the niche.

The above multi-criterion techniques are based on a genetic algorithm. Another approach is an extension of evolution strategy (Kursawe, 1991). Binh and Korn (1997) have developed a multicriteria evolution strategy for combinatorial optimisation problems. A new Pareto archived evolution strategy called PAES has been recently proposed by Knowles and Corne (2000).

Zietzler and Thiele (1998) have suggested an elitist selection in their strength Pareto evolutionary algorithm SPEA. At each generation, a combined population with the external and the current population is constructed. In an external population, all non-dominated solutions discovered so far are archived. An elitist selection is applied in a fast elitist non-dominated sorting genetic algorithm NSGA-II (Deb, Agrawal, Pratap, and Meyarivan, 2000).

6. ADAPTIVE MULTI-CRITERION EVOLUTIONARY ALGORITHM

Figure 3 shows a constructed flow scheme of the adaptive multi-criterion evolutionary algorithm called AMEA. This algorithm achieves better results than the other multiobjective evolutionary algorithms.

The preliminary population is created in a specific manner (Figure 2, line 3). Generated individuals satisfy constraints $\sum_{i=1}^{I} x_{vi}^{m} = 1, v = \overline{1,V}$; $\sum_{j=1}^{J} x_{ij}^{\pi} = 1, i = \overline{1,I}$ by introducing integer representation of chromosomes, as follows:

$$X = (X_1^m,...,X_v^m,...,X_V^m, X_1^\pi,...,X_i^\pi,...,X_J^\pi) \tag{8}$$

where $X_v^m = i$ for $x_{vi}^m = 1$ and $X_i^\pi = j$ for $x_{ij}^\pi = 1$. Furthermore, we assume that $1 \le X_v^m \le I$ and $1 \le X_i^\pi \le J$.

If x is admissible, then the fitness function value (Figure 3, line 4) is estimated, as below:

$$f(x) = r_{\max} - r(x) + P_{\max} + 1 \tag{9}$$

where $r(x)$ denotes the rank of an admissible solution, $1 \le r(x) \le r_{max}$.

1. BEGIN
2. t:=0, set the even size of population L, p_m:=1/M, M – the length of solution
3. generate initial population $\boldsymbol{P}(t)$
4. calculate ranks $r(x)$ and fitness $f(x)$, $x \in \boldsymbol{P}(t)$
5. *finish*:=FALSE
6. WHILE NOT *finish* DO
7. BEGIN /* new population */
8. t:= t+1, $\boldsymbol{P}(t) := \varnothing$
9. calculate selection probabilities $p_s(x)$, $x \in \boldsymbol{P}(t-1)$
10. FOR $L/2$ DO
11. BEGIN /* reproduction cycle */
12. 2WT selection of a potential parent pair (**a**,**b**) from the population $\boldsymbol{P}(t\text{-}1)$
13. S-crossover of a parent pair (**a**,**b**) with the adaptive crossover rate p_c, $p_c := e^{-t/T_{\max}}$
14. S-mutation of an offspring pair (**a'**,**b'**) with the mutation rate p_m
15. $\boldsymbol{P}(t) := \boldsymbol{P}(t) \cup$ (**a'**,**b'**}
16. END
17. calculate ranks $r(x)$ and fitness $f(x)$, $x \in \boldsymbol{P}(t)$
18. IF ($\boldsymbol{P}(t)$ converges OR $t \ge T_{max}$) THEN *finish*:=TRUE
19. END
20. END

Figure 3. Adaptive multicriteria evolutionary algorithm

In the two-weight tournament selection (Figure 3, line 12), the roulette rule is carried out twice. If two potential parents (**a, b**) are admissible, then a dominated individual is eliminated. If two solutions non-dominate each other, then they are accepted. If potential parents (**a, b**) are non-admissible, then an alternative with the smaller penalty is selected.

The fitness sharing technique can be substituted by the adaptive changing of main parameters. The quality of attained solutions increases in optimisation problems with one criterion, if the crossover probability and the mutation rate are changed in an adaptive way proposed by Sheble and Britting (1995). The crossover point is randomly chosen for the chromosome X in the S-crossover operator (Figure 3, line 13). The crossover probability is one at the initial population and each pair of potential parents is obligatory taken for the crossover procedure.

A crossover operation supports the finding of a high-quality solution area in the search space. It is important in the early search stage. If the number of generation t increases, the crossover probability decreases according to the formula $p_c=\exp(-t/T_{max})$. The search region or some search areas are identified after several crossover operations on parent pairs. That is why, value p_c is smaller and it is equal to 0.6065, if t =100 for maximum number of population T_{max}=200. The final smallest value p_c is 0.3679. A crossover probability decreases from 1 to $exp(-1)$, exponentially.

In S-mutation (Figure 3, line 14), the random swap of the integer value by another one from a feasible discrete set is applied. If the gene X_v^m is randomly taken for mutation, the positive integer value is taken from the set $\{1,...,I\}$. If the gene X_l^π is randomly chosen, then the value is selected from the set $\{1,...,J\}$. A mutation rate is constant in the AMEA and it is equal to $1/M$, where M represents the number of decision variables.

An approach based on the genetic programming for solving multicriteria optimisation problems produces a population of algorithms that adapt themselves to the problem. The name "adaptive evolutionary algorithm" for evolutionary algorithms is related with the changing of some parameters as a crossover probability, a mutation rate, a population size, and the others during the searching. For considered algorithm the crossover probability is decreased according to the number of new generations.

7. CONVERGENCE TO PARETO FRONT

The AMEA can be used for solving several multiobjective optimisation problems, if the subset of P-optimal solutions is searched. The AMEA can be applied principally for the problem (4).

The level of convergence to the Pareto front measures the quality of obtained set of solutions by algorithms. It is a closeness measure for the obtained efficient points to the given Pareto points $\{P_1, P_2,..., P_U\}$. Let the AMEA be the set of sub-efficient points $\{A_1, A_2,..., A_{U'}\}$, where $U' \le U$. If there is the outcome $A_u=(A_{u1}, P_{u2}, A_{u3})$ with the same cost of computers as the uth Pareto result $P_U=(P_{u1}, P_{u2}, P_{u3})$, then the distance between these points is equal to $S_{lu}=\sqrt{(P_{u1}-A_{u1})^2+(P_{u3}-A_{u3})^2}$ for the lth run of an algorithm.

If the point A_u is not found by the algorithm, then we assume the distance is $\sqrt{\sum_{\substack{i=1\\ i\neq 2}}^{3}\left(P_{ui}-A_{ui}^{-}\right)^2}$,

where A_{u1}^{-} is the maximal load of the bottleneck computer, and A_{u3}^{-} is the minimal performance of computers, for instance problem (4). ξ initial populations are generated and ξU values of the distances S_{lu} are calculated. Then, a level of convergence to the Pareto front is calculated, as follows:

$$S=\frac{1}{\xi}\sum_{l=1}^{\xi}\sum_{u=1}^{U}S_{lu} \qquad (10)$$

8. BENCHMARK INSTANCE

Several instances of problem (4) have been solved by the AMEA. The number of modules has been changed from three to 50. Let the instance be considered with 10 modules, two nodes, and five computer types. It induces 30 binary decision variables and 1,073,741,824 binary module assignments. Z_{max} is a value from [26; 75] [time unit], F_2 is from [2, 10] [money unit], and $\tilde{F}_2$ from [200, 600] [Mflops].

The number of admissible solutions has an upper bound that is equal to $I^{V}J^{I}$. Figure 4 shows the set of assignment evaluations for this instance. It was found by the exhausted searching of the search space. The point y^{o} is an ideal point for the presented space with the best possible coordinates. It is not included to the criterion space as well as an anti-ideal point y^{-} with the worst possible coordinates. Small circles represent efficient points.

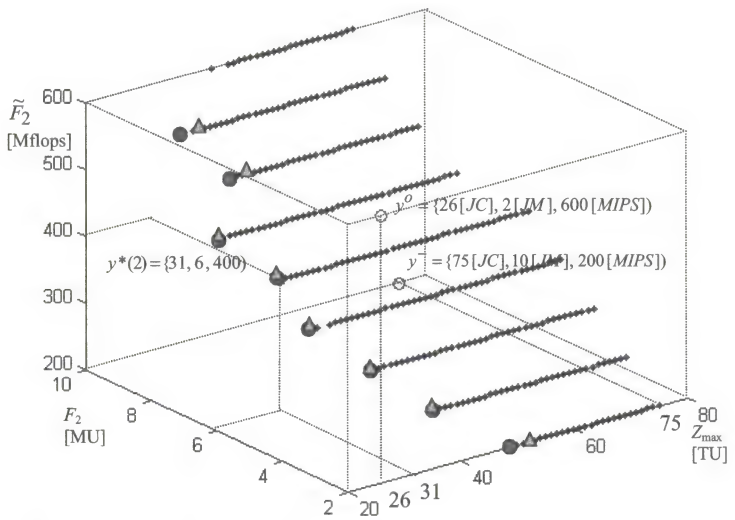


Figure 4. An example of the module assignment evaluation set

The adaptive evolutionary algorithm gives better results than the standard multicriteria evolutionary algorithm SMEA with a constant crossover rate (Figure 5). After 200 generations, an average level of Pareto set obtaining is 1.6% for the AMEA, 3.1% for the SMEA, and 19% for the TMGA with binary chromosomes. Chromosomes for the SMEA have a structure described by (8). Binary chromosomes for the TMGA have a structure described by x at (1). Thirty test initial populations were prepared and each algorithm starts from these populations.

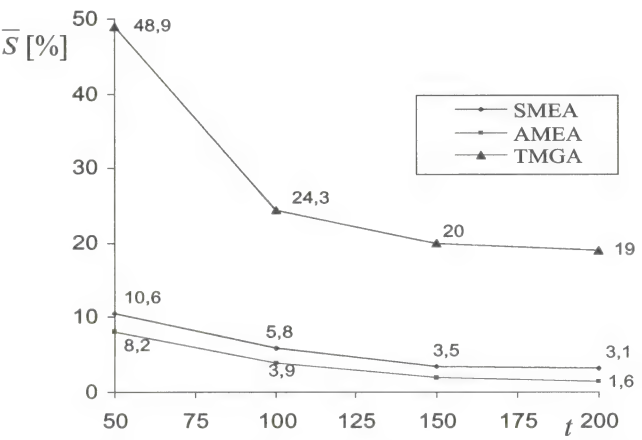


Figure 5. Convergence to Pareto front for some multi-criterion evolutionary algorithms

There are two reasons for improving the quality of obtained solutions. For integer constrained coding of chromosomes, there are 12 decision variables in the test optimisation problem. The crossover operation and the mutation ensure the search space consists of 25,600 solutions.

9. MODULE SCHEDULING

A probability of completing tasks with timing correctness is the fourth measure of module assignment. It can be applied after finding an extended assignment x. For any x, we can calculate a set of probabilities of timely completing a task. Module scheduling is required to find the above probabilities.

It is important to find the worst-case latest completion time of a task. If there are loops or OR-subgraphs in the task flow graph, it can be replaced by a set of component graphs without loops and OR-subgraphs. In the result, we obtain a set of all achievable combinations of probabilistic subgraphs. Each component graph consists of instances of loops and OR-subgraphs. The component task graph $TG(i)$ has the probability of execution p_i.

OR-subgraph is replaced by one of its branch with probability q. If there are N branches, N cases should be considered for each OR-subgraph. A loop is replaced by a cascade of L_{max} copies of a loop body. If a probability for repeating a module in a loop is denoted as l, each kth module in a cascade obtains a probability $(1- l)l^{k-1}$. The kth module bears the latest completion time $(L_{\max} - k)t_{vj}x^m_{vi}x^\pi_{ij}$. There are NL_{max} component task graphs with $p_i=q(1- l)l^{k-1}$ for one QR-subgraph and one loop.

Let the critical time d_v of the vth module be defined as the latest time module that should be completed for the timely completion of the entire task. C_v is the completion time of the vth module. It can be determined by a scheduling algorithm under given extended module allocation x (Weglarz, 1998). A probability of completion N tasks according to several performances represented by the K_n combinations of task flow graph is calculated below:

$$P(x)=\prod_{n=1}^{N}\sum_{i=1}^{K_n} p_i \prod_{m_v\in T_n}\xi(d_v - C_v(x)) \tag{11}$$

where $\xi(d_v - C_v) =1$ for $d_v \geq C_v$ and $\xi(d_v \leq C_v) = 0$, otherwise.

A set $M_i(x)$ of modules belonging to different tasks should be scheduled pre-emptively in each computer. Each module becomes available upon its release at the time. Precedence relations have to be considered in the entire task system to determine the release time for each module. If the vth module precedes the uth module, a successor cannot start its execution before the completion of a parent, regardless of where they are assigned. We assume $v < u$ for this case.

The release time r_v of the vth module belonging to the nth task is at least an invocation time λ_n. r_1 is the invocation time of the task to which m_1 belongs. If only the vth module precedes the uth module, then the release time r_u is calculated, as follows:

$$r_u= r_v+\max\{C_v - r_v,\ t_{vj}x^\pi_{ij}\}+\tau_{vuik}x^m_{vi}x^m_{uk}, u=\overline{1,V} \tag{12}$$

Let r_u be initially set up to the invocation time of the task to which uth module belongs. If a set of modules precedes the uth module, then the release time r_u is calculated, as follows:

$$r_u= \max\{\ r_u,\ \max_{v=1,V}\{r_v+\max\{C_v - r_v,\ t_{vj}x^\pi_{ij}\}+\tau_{vuik}x^m_{vi}x^m_{uk}\ , m_v\rightarrow m_u\}\},\quad u=\overline{1,V} \tag{13}$$

Let d_v be the latest time, when m_v must be completed to ensure the timeliness of succeeding modules. The latest time d_v of the vth module belonging to the nth task is no greater than the task deadline D_n. Let d_v be initially set to the deadline of the task to which vth module belongs. So, d_v is equal to D_n the deadline of the nth task to which m_V belongs. The latest time d_v of the vth module can be calculated, as follows:

$$d_v = \min\{ d_v, \min_{u=\overline{1,V}} \{d_u - t_{vj}x_{ij}^{\pi} - \tau_{vuik}x_{vi}^{m}x_{uk}^{m}, m_v \rightarrow m_u\}\}, \quad v = \overline{V-1,1} \tag{14}$$

We want to find a schedule for the modules in a set $M_i(x)$ that minimises the following objective function:

$$f_{\max}(M_i(x)) = \max_{m_v \in M_i(x)} (C_v - d_v) \tag{15}$$

Module scheduling problem (15) can be solved by several advanced optimisation techniques (Weglarz, 1998). Especially, a module-scheduling algorithm that is suitable for (15) has been considered by Hou and Shin (1997).

10. APPLICATION

In a maritime rescue system, a task set with precedence constraints has been applied for modelling of program modules written in the programming language C (Wesolowski and Zieliñski, 2000). These programs support some navigation tasks on the ship. Moreover, several devices such as the UKF receiver, the GPS, and the digital video recorder are handled by a distributed system. Modules communicate to each other by messages, proxies, and signals. These modules are executed on four computers. Modules communicate by the local area network Ethernet connecting computers equipped with the QNX operating system.

The QNX operating system provides multitasking, priority-driven pre-emptive scheduling, and fast context switching (Krten, 2000). The kernel of QNX delivers messages for modules on other computers. Virtual circuits contribute to efficient communication as well as to use of resources in a network. The virtual circuit handles messages up to a constant size and it is resized to accommodate the larger message. If a module accesses several files on a remote file system, only one virtual circuit exists between this module and files.

Scheduling applies when two or more tasks are ready for using the CPU and are assigned the same priority. If a higher priority process becomes ready, it immediately preempts all lower-priority tasks. There are three scheduling methods for QNX: FIFO, round-robin, and adaptive scheduling. In FIFO scheduling, a selected process continues executing until it is preempted by a higher priority task or if it is blocked during communication. No process will be preempted by the other with the same priority while it is executing. If two modules share a memory segment, each of them could update this segment without resorting to some form of semaphoring. The FIFO scheduling has been applied for the distributed navigation system.

A proposed evolutionary algorithm AMEA has been developed for finding the Pareto-optimal task assignment. Then, the module scheduling has been applied. It has improved several important parameters of an entire system such as the workload of a bottleneck computer, cost of installed computers, a numerical performance of the system, and a probability of completing tasks with timing correctness. The workload of the bottleneck computer has been reduced from 75 seconds to 31 seconds by splitting 30 modules from nine tasks on four computers. Moreover, the other Pareto-optimal task assignment has been found by the AMEA. Also it has a workload of 25 seconds, the other parameters were much worse than parameters of the previous task assignment.

11. CONCLUDING REMARKS

The adaptive evolutionary algorithm AMEA is capable of techniques for solving a new multiobjective optimisation problem focused on finding software task allocations that minimise the workload of the bottleneck computer, the cost of machines, and maximise a performance of the distributed system. It is applied for finding the subset of Pareto-optimal solutions. Then, a module scheduling is applied to maximise a probability of completing tasks with timing correctness for a chosen Pareto-optimal assignment. The practical utility of applying the AMEA algorithm consists in improving main parameters of distributed computer systems.

Our future works will focus on finding the combination, the multicriteria evolutionary algorithm with the tabu search algorithm to include some ranking procedures to the tabu approach for improving the obtained Pareto-optimal task assignments. Another research goal is the extension of the optimisation problem by the introduction of some additional aspects related to including a reliability of computers and communication links to an optimisation model.

REFERENCES

BALICKI, J. (1999): Evolutionary neural networks for solving multiobjective optimisation problems. In *Computational Intelligence and Applications*. SZCZEPANIAK, S. (ed). Heidelberg-New York, A Springer Verlag Company:108-118.

BALICKI, J. and KITOWSKI, Z. (1999): Genetic and evolutionary algorithms for finding efficient program module allocations. In *Advances in Intelligent Systems and Computer Science*. MASTORAKIS N. E. (ed). Danvers, World Scientific and Engineering Society Press:151-156.

BINH, T. T. and KORN, U. (1997): Multiobjective evolution strategy for constrained optimisation problems. *Proc. the 15th IMACS World Congress on Scientific Computation*, Berlin:357-362.

BOKHARI, S. H. (1987): Assignment problems in parallel and distributed computing, Boston, Kluwer Academic Publishers.

CHOI, S. and WU, C. (1998): Partitioning and allocation of objects in heterogeneous distributed environments using the niched Pareto genetic algorithm. *Proc. the 1998 Asia Pacific Software Engineering Conference*, Taiwan:223-234.

CHU, W. W. and LAN, L. M. T. (1987): Task allocation and precedence relations for distributed real-time systems. *IEEE Transactions on Computers* 36(6):667-679.

DEB, K., AGRAWAL, S., PRATAP, A. and MEYARIVAN, T. (2000): A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. Technical Raport No. 200001, Indian Institute of Technology, Kanpur.

FONSECA, C. M. and FLEMING, P. J. (1995): An overview of evolutionary algorithms in multiobjective optimisation. *Evolutionary Computation* 3(1):1-16.

FOURMAN, M. P. (1985): Compaction of symbolic layout using genetic algorithms. *Proc. the First International Conference on Genetic Algorithms*, Hilsdale:141-153.

GOLDBERG, D. E. (1989): *Genetic algorithms in search, optimisation, and machine learning*. Massachusetts, Addison-Wesley Publishing Company, Inc. Reading.

HANSEN, M. P. (1997): Tabu search in multiobjective optimisation: MOTS. *Proc. the MCDM'97*, Cape Town, South Africa.

HOU, C. J. and SHIN, K. G. (1997): Allocation of periodic task modules with precedence and deadline constraints in distributed real - time systems. *IEEE Transactions on Computers* 45(12):1338-1356.

JASZKIEWICZ, A. (2000): On the computational effectiveness of multiple objective metaheuristics. *Proc. the Fourth International Conference on Multi-Objective Programming and Goal Programming*, Berlin-Heidelberg.

KAFIL, M. and AHMAD, I. (1998): Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency* 6(3):42-51.

KNOWLES, J. and CORNE, D. W. (2000): Approximating the nondominated front using the Pareto archived evolution strategy. *Evolutionary Computation* 8(2):149-172,

KRTEN, R. (1998): *Getting Started with QNX 4. A Guide for Realtime Programmers*. PARSE Software Devices,.

KURSAWE, E.(1991): A variant of evolution strategies for vector optimisation. In *Parallel problem solving from nature*. SCHWEFEL, H.-P. and MANNER, R. (1991) eds. Lecture Notes in Computer Science 496:193-197.

LO, V. M. (1988): Heuristic algorithms for task assignments in distributed systems. *IEEE Transactions on Computers* 37(1):1384-1397.

MICHALEWICZ, Z. (1996): *Genetic Algorithms + Data Structures = Evolutionary Programs*, Springer Verlag.

SCHAFFER, J. D. (1985): Multiple objective optimisations with vector evaluated genetic algorithm. Proc. the International Conference on Genetic Algorithms and Their Applications, Hillsdale, 93-100.

SHEBLE, G. B. and BRITTING K. (1995): Refined genetic algorithm – economic dispatch example. *IEEE Transactions on Power Systems* 10:(2):117-124.

SINCLAIR J. B. (1988): Optimal assignments in broadcast networks. *IEEE Transactions on Computers* 37(5):521-531.
SRINIVAS, N. and DEB, K. (1994): Multiobjective optimisation using non-dominated sorting in genetic algorithms, *Evolutionary Computation* 2(3):221-248.
STONE, H. S. (1977): Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering* 3(3):85-93.
TOWSLEY, D. F. (1986): Allocating programs containing branches and loops within a multiple processor system. *IEEE Transactions on Software Engineering*, 12(10):1018-1024.
WESOLOWSKI, M. and ZIELINSKI M. (2000): The post of recognition the UKF band. *Proc. The 9th International Conference on Military Communication and Information Systems*, Zegrze, Poland, 3:147-150.
WEGLARZ, J. (1998) (ed.): *Recent advances in project scheduling. Dordrecht*, Kluwer Academic Publishers.
ZITZLER, E., DEB, K. and THIELE L. (2000): Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation* 8(2):173-195.
ZITZLER, E. and THIELE, L. (1998): Multiobjective optimisation using evolutionary algorithms – A comparative case study. In *Parallel Problem Solving from Nature*. EIBEN A. E. *et. al.*, Springer-Verlag, Berlin, 292-301.

BIOGRAPHICAL NOTES

Jerzy Balicki is a lecturer in the Computer Science Department at the Polish Naval Academy. He has published over 100 papers in the area of distribution computation, artificial intelligence and multicriteria optimisation. He received his PhD in computer science at the Military Technical Academy in Warsaw. His projects include multiobjective optimisation of distributed computer system performance by modern heuristic techniques.

NOTES

NOTES

NOTES

NOTES

Submission Guidelines

The Journal encourages submission of innovative and original articles in all areas of Information Technology including Computer Science, Software Engineering, Information Systems, Computer Systems and Information Engineering and Telecommunications.

The following forms of article are published:

- Full articles examining original research and/or application;
- Short articles describing original research and/or application or commenting on relevant legal, political or technical innovations;
- Survey or tutorial articles describing new directions in Information Technology or which are relevant to the practical application of fundamental research;
- Book Reviews;
- Announcements; and
- News Briefs.

Full details can be found at the Journal home page: www.jrpit.acs.org.au

Subscription Guidelines

The annual subscription for the Journal is

Individuals	
Australian Computer Society members	Free
Delivery address in Australia, incl. postage and GST when applicable	A$66
Overseas, incl. postage.	US$54
Institutions	
Australia, including postage and GST where and when applicable.	A$99
Overseas, including postage	US$81

All subscriptions to the Journal are payable in advance and should be sent (in Australian Currency) to the address below.

Advertising Information

The Journal accepts advertisements in the following categories:

- IT-related career opportunities
- IT-oriented training and education courses;
- IT-related conferences and workshops.

Rates are as follows:

Full Page	$1,500
Half Page	$750
Quarter Page	$375
By line	$100 for heading plus up to six lines of text (40 characters per line).
	$50 for each additional three lines, or part thereof.

Full details can be found at the Journal home page http://www.jrpit.acs.org.au

Bibliographic Information

Journal of Research and Practice in Information Technology – *J.Res.Prac.Inf.Tech.*
formerly the Australian Computer Journal (ISSN 004-8917)

ISSN	1443-458X
Circulation	15,500
Publisher	Australian Computer Society Inc. PO Box Q534, QVB Post Office Sydney 1230, New South Wales AUSTRALIA
Production Management	Associated Business Publications Pty. Ltd. P.O. Box 3267, Umina Beach, NSW 2257, Australia
Printer	Mailing and Print Services Pty Ltd 8 Aquatic Place, Frenchs Forest, NSW 2086, Australia
Cover Design	Modern Planet Design (08) 8340 1361 • modern@webmedia.com.au

SPECIAL ISSUE ON DISTRIBUTED SYSTEMS

Contents

107 Performance Evaluation of Middleware Bridging Technologies
R. Fatoohi, V. Gunwani, Q. Wang and C. Zheng

120 An Extensible Framework for Testing Distributed Java Applications
S. Ghosh, N. Bawa, G. Craig and K. Kalgaonkar

133 Using XML to Specify a Trace Format for MPI Programs
S. Huband and C. McDonald

146 Practical Experience Using a Computational Model for the Design of Heterogeneous Distributed Software
T. Williams and R. Parsons

158 RBA: A Best Effort Resource Allocation Algorithm for Asynchronous Real-time Distributed Systems
B. Ravindran and T. Hegazy

173 Multi-criterion Optimisation of Distributed System Performance by Evolutionary Task Assignments
J. Balicki

Special Features

105 Guest Editorial Michael Oudshoorn

A publication of the Australian Computer Society
ISSN 1443- 458X